Some Visualization Models applied to the Analysis of Parallel Applications

Lucas Schnorr

INSTITUT POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

THÈSE EN COTUTELLE INTERNATIONALE

pour obtenir le grade de **DOCTEUR DE L'Institut polytechnique de Grenoble**

Spécialité : Informatique

et de **l'Université Federale du Rio Grande do Sul**

préparée au Laboratoire d'Informatique de Grenoble dans le cadre de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique** et au Laboratoire de Parallelisme et Distribution dans le cadre du **Programme de Doctorat en Informatique**

préparée et soutenue publiquement par

Lucas MELLO SCHNORR

le 26 octobre 2009

Titre :

Some Visualization Models applied to the Analysis of Parallel Applications

---

sous la direction de Denis Trystram, Guillaume Huard et Philippe O. A. Navaux

---

JURY

Jean-François MÉHAUT, Président
Eddy CARON, Rapporteur
Siang SONG, Rapporteur
Nicolas MAILLARD, Examinateur
Denis TRYSTRAM, Directeur administratif
Guillaume HUARD, Directeur de thèse
Philippe O. A. NAVAUX, Directeur de thèse

Acknowledgments

A thesis demands a continuous effort and dedication for a long period of time and space. It is with great happiness that I write these words to thank the many people who made this french-brazilian thesis possible.

In the first place, I would like to thank my advisor Dr. Philippe O. A. Navaux, for accepting me as his Ph.D student back in 2005, for giving me the opportunity to prepare and write this thesis within an international environment, for motivating me during all this time to publish my work and go present them in the conferences, and also for putting me in contact with several different people from the parallel application area in Brazil, France and abroad.

I also would like to thank my second advisor Dr. Guillaume Huard, for his comments, suggestions, and ideas that inspired me to prepare this thesis. Together, we defined the first sketches of this thesis during the beginning of my stay in France, which enabled the creation of good ideas and experiments. I also thank him for his patience during my first months in France, when my french was not that good yet.

I am also grateful to Dr. Denis Trystram, for accepting me as his student in the Grenoble INP, and his availability to guide me every year towards the accomplishment of this thesis.

I would like to thanks the members of my jury: Eddy Caron, Jean-François Méhaut, Nicolas Maillard, and Siang Song for their comments on my text that allowed me to improve the clarity and the quality of this manuscript. Their suggestions for future work also inspire me to continue with my research within the performance visualization area.

This french-brazilian thesis would not have been possible without the financial support of scholarships from CAPES, CAPES/Cofecub, and CNPq. I am also grateful for the different projects in France and Brazil for financing my trips to the different conferences I attended.

I would like to thank all my friends, colleagues from the GPPD group in Brazil and the LIG in France, and family, for their presence and support during all this thesis. Each of you is part of thesis through the several discussions and ideas we had together.

Lastly, and most importantly, I wish to thank my wife, Fabiane P. Basso. I simply do not have words to thank you for loving me all these years. The force of our love is marked in every word of this text, especially during the time when we were apart because of the international nature of my thesis. I will be grateful to you all my life.

Obrigado! Merci!

Contents

Chapter 1

Introduction

Distributed systems are related to hardware and software that contain more than one single processor entity [19]. In such systems, processors are interconnected and communicate over a network. The computer programs that execute in these systems are split into multiple parts and must deal with different levels of parallelism and with communication paradigms, such as message-passing and shared memory. A kind of distributed systems is grids [30]. They are often structured in virtual organizations [29], possibly composed by thousands of machines distributed geographically. Two examples of this type of system are the french Grid'5000 [12] and the american TeraGrid [16].

Characteristics shared by almost all grid platforms are dynamism, heterogeneity of resources and software, and presence of multiple administrative domains. Dynamism means that the resources that participate in the grid can be unavailable at any time, without any prior notification of that. Parallel applications must deal with that in the application-level or through a middleware capable of handling resources fluctuations. The heterogeneity means that different configurations of resources can be present in the same grid infrastructure. This is also valid for software libraries and middlewares. A grid can be scattered through multiple administrative domains, each part handled independently by their administrators. Besides these characteristics, a grid might also have a complex network interconnection and be easily extensible in terms of resources.

The interconnection among resources of a grid can be composed of different types of networks. They include Ethernet, Myrinet, Infiniband, or optical fiber technologies. A model of a grid with several types of interconnection is a desktop grid [48], like the projects BOINC [1] and Seti@home [2], where the network is the internet. Another example for the presence of multiple types of interconnection is a lightweight grid, where a strong hierarchy is used to interconnect a set of homogeneous clusters of computers [12]. The presence of several interconnections come from the natural heterogeneity and geographic distribution characteristics of grids. These aspects impose a higher network complexity, greater number of hops to provide communication among applications processes, and increasing differences in network latencies and bandwidth.

Grid platforms are also easily extensible. New resources can be indefinitely added just by connecting them to the existing participants. Usually, these additions bring more heterogeneity to the grid and increment the network complexity. As of today, there are global grids that are

composed of several thousands of computers, such as the example of BOINC. Another example of how easy it is to add new resources to a grid is the case of Grid'5000, where new clusters and sites can be added to the main backbone of the infrastructure. The extensibility of these grid platforms is a good characteristic from the point of view of parallel applications, which need an increasing amount of resources to complete their execution faster.

All these grid characteristics influence directly the behavior of the parallel applications during their development and execution. Because of this, it is important for the developer to understand the impact of the distributed system on the application. An example of this type of analysis is the observation of application monitoring data with information from the network topology. The application can have a better or worse performance, depending on which resources are chosen and the interconnection among them. This influence is even more evident when network aspects are considered, such as latency or bandwidth, on network-bound parallel applications. The grid extensibility is another aspect that influences directly the behavior of applications, because turning available new resources for the application might not always result in a better performance.

Considering these situations, we can notice that it is important to analyze the parallel application behavior along with information about the grid resources. This analysis can help the developer to understand the impact of the network topology on the application behavior. Comparing, for instance, the communication pattern of the application with the network topology can give hints to the developer to adapt the application in order to better exploit the interconnection. Moreover, if the network is hierarchically organized, the applications can follow the hierarchy to avoid bottlenecks.A good analysis must also lead to conclusions about all processes of parallel applications, including global and local patterns that can appear among them. If the number of processes is large, the analysis must also scale.

Visualization is a way to perform the analysis of parallel applications. It has been extensively used through the last 30 years to understand and observe applications that are developed with different levels of parallelism. The most traditional way of visualizing application behavior is through an adaptation of Gantt charts [79], also known as space-time graphics. They list the components of the application vertically and their evolution over time is placed on the horizontal axis. Examples that provide this kind of visualization are Pajé [22], Vampir [60] and many others [5, 46, 63]. This visualization is already widely used in existing architectures, such as clusters, where data is simpler and uniform.

Many of these tools were adapted to observe the behavior of applications of highly distributed systems like grids. Generally, they keep on using the same visualization techniques. Considering only the space-time representation, the first issue that arises is that they cannot represent, together with the application data, the complex network topology of grid systems. As discussed, the impact of the network cannot be excluded from an application analysis when a complicated interconnection is present among the resources. The second problem is the visualization scalability of the space-time approach. Using such representations, the number of components of the application that can be visualized in a screen is limited to the vertical resolution of the screen.

This thesis tries to overcome the issues encountered on traditional visualization techniques for parallel applications. The main idea behind our efforts is to explore techniques from the

information visualization research area and to apply them in the context of parallel applications analysis. Based on this main idea, the thesis proposes two visualization models: the three-dimensional and the visual aggregation model. The former might be used to analyze parallel applications taking into account the network topology of the resources. The visualization itself is composed of three dimensions, where two of them are used to render the topology and the third is used to represent time. The later model can be used to analyze parallel applications composed of several thousands of processes. It uses hierarchical organization of monitoring data and an information visualization technique called Treemap [74] to represent that hierarchy. Both models represent a novel way to visualize the behavior of parallel applications, since they are conceived considering large-scale and complex distributed systems, such as grids.

The implications of this thesis are directly related to the analysis and understanding of parallel applications executed in distributed systems. It enhances the comprehension of patterns in communication among processes and improves the possibility of matching them with real network topology of grids. Although we extensively use the network topology example, the approach could be adapted with almost no changes to the interconnection provided by a middleware of a logical interconnection. With our scalable visualization technique, developers are able to look for patterns and observe the behavior of large-scale applications.

In this work, we are considering parallel applications that intend to obtain high performance in grid environments. Additionally, these applications must be composed of processes that intercommunicate during the execution of the application, either as point-to-point communications or collective operations. Each process is composed of functions related to calculations or to communicate with other processes. Besides this, we also consider that the number of processes of the same application can scale up to a large number. To analyze these applications, we consider that traces can be generated during application execution. A trace is divided in timestamped events, each one identified by a type and additional information according to this type. Several types of events might be registered, for instance, the start and end of functions, the communications, and so on.

The text of the thesis is composed of six chapters, as follows:

Chapter 2: Visualization of Parallel Applications

This Chapter presents works related to this thesis. It starts with a historical presentation of tools since their first use to analyze computer programs, then goes to the description of some of them. The Chapter ends with a summary of visualization techniques, classified according to three types according to the information they represent.

Chapter 3: The Three-Dimensional Model

This Chapter presents the three dimensional model. We first describe its visual conception, detailing the components and concepts of the 3D visualization. Afterwards, we describe the abstract model that is conceived to generate these visualizations. During this description, we detail three different cases that can be rendered with the approach to help the performance analysis of parallel applications.

Chapter 4: Visual Aggregation Model

The fourth Chapter presents the visual aggregation model, proposed in this thesis to be

combined with the treemap representation so the analysis of parallel applications can be done with a large number of components. The Chapter first details how monitoring data can be hierarchically organized, then it goes through the description of the proposed Time-Slice algorithm and the aggregation model. The Chapter ends with the use of the treemap technique to visualize the hierarchies created by the proposed algorithms.

Chapter 5: Triva Prototype Implementation

The fifth Chapter presents Triva, a prototype that includes the implementation of the three-dimensional and the visual aggregation model. A performance evaluation of some Pajé components is included in the beginning of this Chapter, in order to introduce the use of these components inside Triva. The rest of the Chapter presents the implementation decisions and the description of the several modules, such as the DIMVisualReader, to read traces, the TrivaView, to the 3D views, and the TimeSliceView, related to the aggregation model.

Chapter 6: Results and Evaluation

The sixth Chapter presents the results obtained with the Triva prototype and its evaluation, through a set of synthetic and real scenarios that lists the main benefits of the proposed approaches. A traces description is given in the beginning, detailing the synthetic, KAAPI and MPI traces used in the experiments and how they were obtained. Then, we present the resulting 3D visualizations rendered by the prototype and finish the Chapter with the presentation of several treemaps whose hierarchies were created by the Time-Slice and aggregation algorithm.

Chapter 7: Conclusion

The main contributions of this thesis are reminded and the perspectives that are opened by its concepts are detailed.

Chapter 2

Visualization of Parallel Applications

The main objective of the performance analysis of programs is to improve the behavior of applications. This analysis is more complex in a parallel or distributed execution environment, since there is a large number of variables that influence the execution of the applications. Common problems are network contentions, bottlenecks, dead-locks, and so on.

The performance visualization of parallel applications is an alternative to perform the analysis. It explores graphical representations and techniques to represent the application behavior. A lot of efforts have been applied in the development of new visualization schemes and techniques in the last 25 years. Most of this development is focused in the adaptation of the visualization techniques to new programming paradigms and libraries for parallel applications. An example of that is the appearance of the MPI Standard, in the middle 90's, and the development of large-scale clusters. In this Chapter we present the techniques and tools that contribute to the area of performance visualization of parallel applications.

The Chapter is organized as follows. We start by describing the evolution of performance visualization tools in Section 2.1, including a correlation between the tools and their creators. In Section 2.2, we detail a representative set of these tools, based on the innovative visualizations they provided when they were published. The Chapter ends with a classification of the visualization techniques, in Section 2.3, and a summary of the Chapter.

2.1 Historical Evolution

The history of visualization tools for program analysis is closely related to the first successful appearance of graphical user interfaces in 1984, with the release of the Macintosh, by Apple. With a wider availability, graphical interfaces have begun to be explored by a series of projects in the United States, almost at the same time. Figure 2.1 depicts a timeline view of a selected set of visualization tools for parallel program analysis. The timeline covers almost 25 years, from 1985 up to now. The year associated with each visualization tool is only an approximation based on publications and technical reports.

The first known project that discusses the possibility of using graphical analysis for the comprehension of parallel programs is the Programming and Instrumentation Environment for Parallel Processing – PIE [71], developed at the University Carnegie-Mellon. Although the

project has started in 1985, first results showing a complete use of visualization techniques of PIE have appeared only in 1989 [51]. The IPS [58, 82] proposes, in 1987, a hierarchical model for constructing parallel applications. Its second generation [56] features an interactive user interface with graphics showing resources metrics that were registered during program execution. On top of the hierarchical model proposed by IPS, the second generation presents graphics with different hierarchy levels, such as machines, processes and threads.

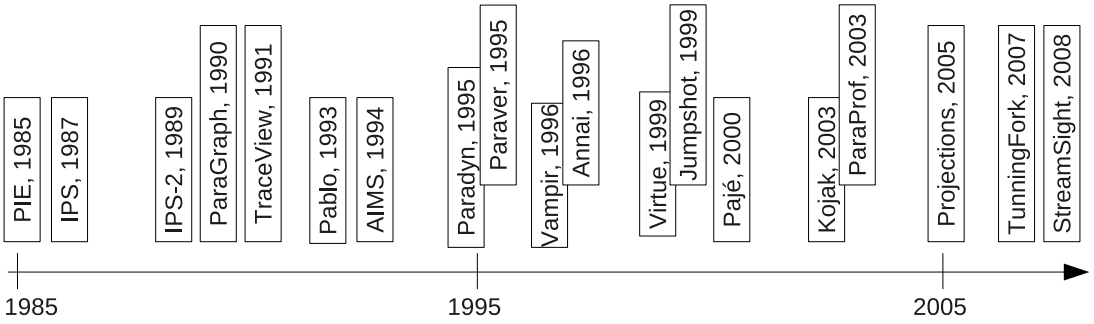


Figure 2.1 – Timeline of significant visualization tools for parallel program analysis.

ParaGraph [38], initially developed at the University of Illinois, is a software that provides a detailed, dynamic, graphical animation of the behavior of message-passing parallel programs and graphical summaries of their performance. At least 25 different types of views are available for the developer to understand the application behavior. Their authors were the first to use the term “simulation” to mean graphical animation, stating that there is nothing artificial about the analysis, but that the behavior of the application is reconstructed with a simulation based on real trace data. Its implementation uses the Portable Instrumented Communication Library – PICL [36] as data source. Because of this dependence on PICL, the tool was considered limited since it was not possible to analyze other types of parallel applications, such as the ones with multiple threads or a combination of message-passing and threads.

The first effort in direction of a more general-purpose tool appears in TraceView [53]. The notion of a general-purpose tool was developed to avoid a particular trace format, a specific execution paradigm or execution system. According to the authors of TraceView, the architecture of the tool is flexible enough to select different analysis and display alternatives, but rigid enough to combine these alternatives based on the resources of the tool.

The evolution of parallel computer systems and larger applications presented new challenges in terms of performance visualization. The first tool to address this issue is Pablo [66]. The tool is built as a series of interconnected components. As trace data moves through these components, it is transformed in a way to provide different views. The development of the tool brings the proposal of SDDF [6], a self-describing trace format.

AIMS, for Automated Instrumentation and Monitoring System, is a toolkit developed at NASA in 1994 to facilitate the performance evaluation of parallel applications via measurement and visualization of execution traces [81]. It has four main components: a source-code instrumentor; a run-time performance monitoring library; two tracefile analysis tools and a trace post-processor to compensate the intrusion caused by the tool in the application execution.

The main characteristic of Paradyn [57] is the Performance's Consultant that helps the developer to dynamically set instrumentation points in the parallel programs. By doing this, the authors argue to improve scalability by reducing intrusion problems during application execution. Paraver [63] also appears in 1995 and offers the possibility to choose different filters to select what is going to be displayed.

Vampir [60], by Pallas GmbH, is a commercial visualization tool for the analysis of parallel applications following the MPI standard. It offers to the developers a wide range of graphical views, such as state diagrams, activity charts, timeline displays and so on. It also has flexible filter operations to reduce the amount of information displayed. The tool has been improved with techniques such as the hierarchical visualization in time-space diagrams [14] to handle large applications.

Annai [18] is an integrated environment for performance visualization of applications developed with High Performance Fortran and with MPI.

In 1999, Virtue [72] brings to the performance visualization new concepts where human sensory capabilities are explored with a 3D immersive visualization. At the same time, the development of MPI results in the first Jumpshot visualization tool [83], developed in Java. Jumpshot is the evolution of the first MPI analysis tool of the same team. The new version contains a number of enhancements in order to make it suitable for large-scale analysis. Jumpshot is still in development and is now in its fourth version. The general purpose visualization tool Pajé [76], presented in 2000, proposes a file format without semantic and strongly related to visual objects. The tool is extensible, interactive and scalable, being capable to visualize any kind of monitoring data that can be described in its format. Kojak [59] appears in 2003 and is developed by the Julich Supercomputing Center in Germany. It supports programming models such as MPI, OpenMP, Shared memory and combinations of them. Its main idea is the automatic search of event traces that indicate inefficient behavior. The results are presented with a graphical user interface. Also in 2003, the ParaProf [10] is presented as a portable, extensible and scalable tool for parallel performance profile analysis. The idea of Paraprof is to gather in the same tool the best capabilities from all previous performance analysis tools. The Projections tool [44], introduced as a preliminary study in 1992, but only available around 2005, is developed to visualize the behavior of Charm++ [45] parallel applications. It has multiple views and techniques to reduce the amount of trace data.

More recently, in 2007, the TuningFork [7] proposes visualization techniques to analyze large-scale real-time systems. Although not directly related to the analysis of parallel applications, many of the problems faced by TunningFork are the same of traditional parallel applications. Examples of these problems are trace collection, very large traces analysis, vertical integration of data, and so on. Another tool is StreamSight [25], a tool developed to understand the dynamic behavior of streaming applications. It has the ability to visualize applications with thousands of nodes and interconnections.

As a conclusion, we can notice that the first tools were mainly focused in the way applications should be instrumented. Dynamic and automatic instrumentation techniques were also proposed. Then, the focus moved to more general and modular tools that are extensible to other programming paradigms. The visualization techniques evolved rapidly in the beginning and are continuously explored till today. Recent tools try to solve the problem of visualizing enor-

mous amount of data, acting directly with reducing and aggregation mechanisms or with new visualization schemes that support more data to be represented.

Mapping Tools to Authors

The timeline evolution of performance visualization tools, together with their respective authors, can be used to analyze how the research area has evolved in the last 25 years. Figure 2.2 shows a mapping between performance visualization tools and their authors. Some authors created more than one tool over time, improving the area of performance visualization analysis. An example is Barton P. Miller, who has worked in the IPS project and is active today working in the same area, with the Paradyn tool. Another author that is still active in the research area is Allen D. Malony, who in 1991 proposed TraceView and currently is working in the TAU's ParaProf performance visualization tool.

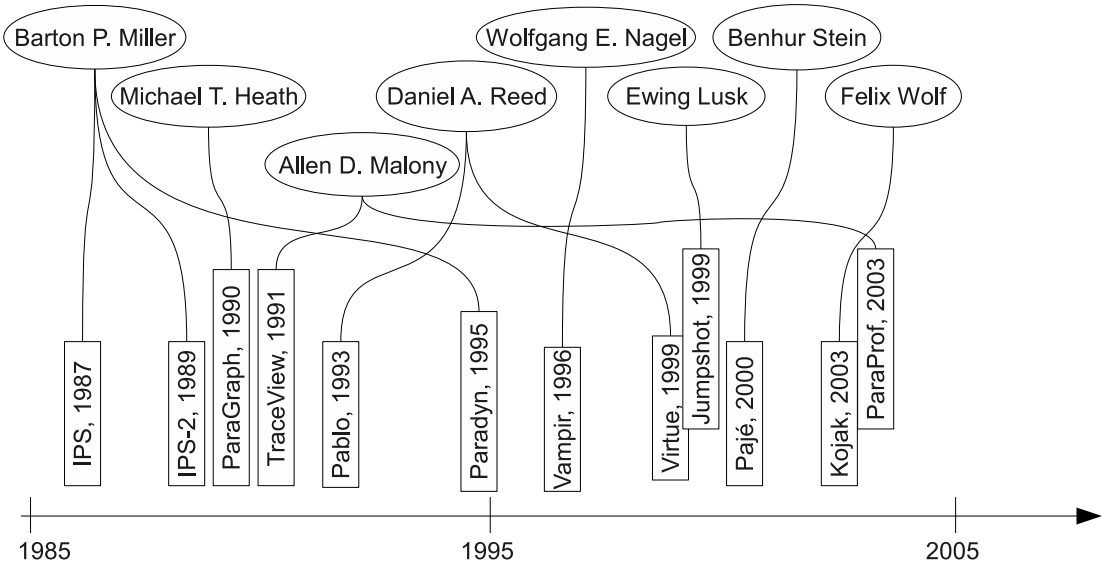


Figure 2.2 – A mapping between performance visualization tools and their authors.

Another possible analysis of Figure 2.2 is to check where the tools and their ideas have been proposed. Up to 1995, all performance tools of the Figure came from two places: the University of Wisconsin, Madison (as it is the case of Barton P. Miller, after finishing his Ph.D. at the University of California, Berkeley) and the University of Illinois at Urbana-Champaign (Michael T. Heath, Allen D. Malony and Daniel A. Reed). After 1994, with the definition of the MPI Standard at the Argonne National Laboratory (ANL), the area of performance visualization starts to be explored in other places: Vampir in 1996 and Kojak in 2003, in the Julich Supercomputing Center in Germany; Jumpshot in 1999, at the Argonne National Laboratory in the United States; Pajé in 2000, in the ID Laboratory, France and at the Federal University of Santa Maria, Brazil, for example.

2.2 Some Performance Visualization Tools

A lot of efforts have already been made in the performance visualization area by different research projects. These efforts resulted in a considerable amount of visualization techniques, from specific tools attached to a programming paradigm, to more generic or evolutive tools that have been adapting to new challenges and evolutions of the high performance domain.

The positive side of specific tools is the number of users that increases rapidly, since they do not need to learn too much to use them. Their main drawback, however, is that they might become obsolete shortly. This is usually caused by a new parallel programming paradigm that cannot be represented in the tool, or by scalability issues, when the tool is no longer able to handle an increasing amount of monitored entities for instance. On the other side, generic or evolutive tools live longer, but their use stays limited because users must continuously learn to keep up with their changes, or must spend more initial effort in learning how to use them.

We present here some performance visualization tools that were developed by different performance research groups. Although the list of tools we describe here is not exhaustive, we think that they represent well the state of the art of the area of performance visualization. Some of them are no longer supported, such as ParaGraph, TraceView and Pablo. Some are still under development and available for the community. For all of them, we present the more relevant ideas, especially the ones related to visualization techniques.

2.2.1 ParaGraph

ParaGraph [38] was initially developed at the Oak Ridge National Laboratory, in Tennessee, United States. Afterwards, ParaGraph started to be hosted and developed at the Center for Simulation of Advanced Rockets, at Urbana-Champaign.

The tool is the first to use the term simulation during the creation of a visual representation of trace data. The term is used because the tool has to re-create the behavior of the application based on real events collected during the parallel application execution. This behavior is then visualized through different visualization techniques, some of them illustrated in Figure 2.3. The first implementation of ParaGraph was able to visualize only message-passing parallel programs developed with the PICL [36] communication library, through the use of specific functions that exchange messages among processes. In the beginning, this coupling between ParaGraph and PICL was seen as positive, because the cycle of development, execution and analysis was straightforward. However, as new communication libraries have started to appear with better performance, the coupling between ParaGraph and PICL became a limitation, because they were attached to a specific communication library. After the appearance of the Message-Passing Interface (MPI) specification [37], around 1994, the PICL evolved with a new trace format and it is renamed to MPICL, addressing parallel applications developed following the MPI specification.

The architecture of ParaGraph is based on events. The visual representations are updated as new events are read from the trace files. The tool is also considered as an interactive interface, the user has access to more than 25 displays, categorized in three families: utilization, communication and tasks. If the user decides to visualize more than one display at the same time, they are kept synchronized. Besides that, the limit for visualization of most displays is 512 processors.

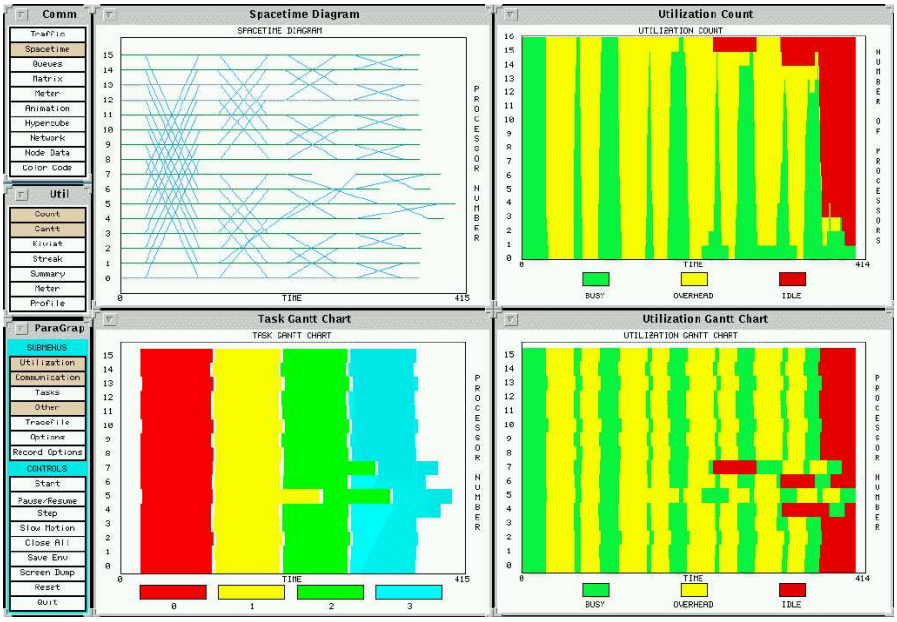


Figure 2.3 – Four different visualizations of ParaGraph.

The **utilization family** is composed by 7 displays: the utilization count, showing the total number of processes in each of three states (busy, overhead and idle); the Gantt Chart, showing the activity of individual processors through time; the Kiviat Diagram, that gives a geometric representation of the utilization of individual processors and the load balance across all processors; the Streak, showing insights of patterns in parallel programs or imbalances among the processors; the Utilization Summary, showing the cumulative percentage of time that each processor spent in each of the three states; the Utilization Meter, that shows the same information as the utilization count, but saving screen space; and the Concurrency Profile, showing the percentage of time that a set of processors remained in the same state.

The **communication family** of displays has 10 different views. The Communication Traffic, showing the total traffic in the communication system as a function of time. The Spacetime Diagram, showing the interactions among processors as a function of time. Message Queues, which is a graph showing the evolution of number of buffer messages through time. Communication Matrix is a two-dimensional array where rows and columns represent processors and each position in the matrix represents a communication between two processors. The Communication Meter uses a vertical bar that indicates the amount of communications in transit (sent but not received). The Animation display shows a graph where the nodes are the processors and the arcs are the communication among them. The nodes can be arranged in pre or user-defined configurations. Hypercube is another display that looks like the animation display, but focused on hypercubes. The Network display shows the path that each message takes to go from one processor to another, including routing through intermediate nodes. This display needs a topology description to be rendered. Node Data presents statistical data in graphical form, such as given variable of the application in function of time. The last one is the Color Code display, helping

to define colors that are used through the other displays. The Animation and Network display of ParaGraph are limited to 128 processors, because of their higher detail level. Hypercube is limited to 16 processors and the Node Data is limited to 256 processors.

Summarizing, ParaGraph's utilization and communication displays only show information about the processors used by the parallel application. The **task family** of displays intend to give developers more insights about the reason behind those behaviors, showing application details. The events shown by these displays must be generated by parallel application developers, through instrumentation of parallel programs. Among the available displays, users have the count, gantt, status, summary displays. They use the same visualization techniques of the communication family of displays, but showing application-level traces.

Besides these three types of displays, ParaGraph has also another set of views that does not fit in one of these types, or fit in more than one type. Among them, there is the Critical Path display, which is a variant of the spacetime display, showing a different color coding to highlight the longest serial thread in the parallel computation. ParaGraph architecture has also the ability to receive new displays to represent in different ways the traces.

The main contribution of ParaGraph is the large set of visualization techniques that could be applied to the same set of traces. Even if applied in a low scale up to 512 processors with some techniques, the visualizations developed in the tool have inspired subsequent tools.

2.2.2 TraceView

TraceView [53] is a trace visualization tool developed at the Center for Supercomputing Research and Development, at Urbana-Champaign, United States. The main idea behind TraceView is to be a general-purpose trace-visualization system. To achieve that, the tool uses the concept of visualization session, defined as a hierarchical structure with three levels: the trace files, the views and the displays. There is also a session management component that helps users to define the specific hierarchical structure needed for the analysis of given set of trace files. TraceView avoids semantic interpretation of the actions registered in the events, meaning that the tool can adapt to different types of traces. In terms of visualization, TraceView offers two types of display, both based on gantt-charts: the Gantt Chart Widget and the Rates Display. The former creates a visualization focused on state transitions of processes; the latter displays the number of times a given state is entered.

As conclusion, TraceView was the first to mention the general-purpose idea in trace visualization systems. The term "general" was used by its authors to mean the way trace files, views and displays should be organized, to build an analysis environment.

2.2.3 Pablo

Pablo [64, 65] is a performance analysis environment designed to provide performance data capture, analysis, and presentation. It is developed at the Department of Computer Science in the University of Illinois at Urbana-Champaign. The tool is conceived to support portability, scalability and extensibility.

The tool is composed of different modules that can be interconnected as a graph. The modules are responsible for data transformations that generate performance metrics for the analysis.

There are modules for operations like selection, arithmetical and logarithm operations, statistical functions and so on. Besides them, Pablo comes with components to read and write trace files. The user of Pablo is then responsible for visually arranging a graph of these modules in order to analyze the traces. All modules developed for Pablo have no semantics, working with any data that is available by the reading modules, independent of what they mean.

Input files of Pablo must conform to the SDDF format [6]. The format is also used internally by the tool. With that, the user can attach to any module an output trace file writer that will write in files the results obtained in the middle of a performance analysis.

In terms of visualization, Pablo offers different techniques to represent the performance data generated by the graph of modules. Basic charts like bar graphs, bubble, strip, and pie charts, contour and interval plots are available for the user by attaching them to the output of a module. Other visual representations, some of them already present in tools such as ParaGraph (see Section 2.2.1), like matrix displays and kiviat diagrams, can also be used. A notable visualization technique, presented at the time of its creation, is the 3-dimensional scatter plot: the technique is used to show, at the same time, three different performance metrics.

Pablo's main contributions are the use of trace files in the SDDF format, and its internal organization in modules, allowing extensions to be made. Its drawback, however, is related to the way these extensions must be developed, since all modules must be integrated in the same binary to make the tool work.

2.2.4 Paradyn

Paradyn [57] is a tool to measure the performance of large-scale parallel applications. It is developed at the University of Wisconsin, Madison, in the United States. The main idea of the tool is to support the dynamic instrumentation of parallel applications in order to be less intrusive and to avoid generating trace data for regions of parallel code that are not under analysis. Paradyn also aims to be scalable, to provide well-defined data abstraction, to support heterogeneous environments and to offer open interfaces for visualization and new data sources.

Perhaps the more interesting idea of Paradyn is the dynamic instrumentation of parallel programs. It works by inserting instrumentation points to detect general high-level performance problems. If a problem is found, Paradyn increases the instrumentation level in those areas that are presenting performance issues. The benefit of this technique of instrumentation is that it decreases the intrusion caused by unnecessary code insertions, with the drawback of being tightly related to the parallel programming paradigm used. This technique is implemented within Paradyn through its Performance Consultant, an implementation of the W3 Search Model [40].

In terms of visualization, Paradyn has a set of pre-defined standard visualizations, like time histograms, and bar graphs. Some examples of these standard views are in Figure 2.4. According to Paradyn's authors, the process of adding new visualizations to the tool is easy because of a special mechanism dedicated to that. The controller of the visualizations runs as a separate process. It can contact Paradyn's main processes to collect data, which is stored in a data structure called a time histogram. Another feature of its visualization system is that Paradyn can incorporate displays from other tools such as ParaGraph and Pablo.

The time histogram visualization of Paradyn plots performance data for metric over time. The horizontal axis represents time and the vertical axis represent the metric that is currently be-

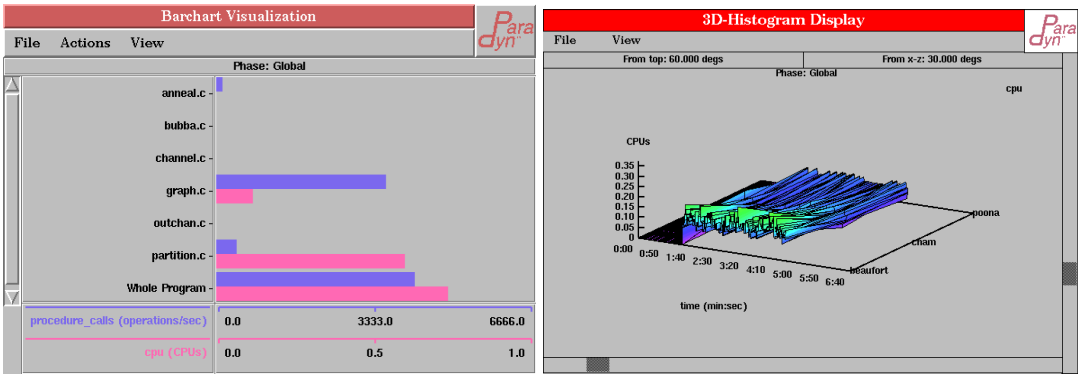


Figure 2.4 – Two visualizations of Paradyn, including the 3D histogram (at right).

ing observed. Several metrics can be analyzed at the same time, and in this case, the vertical axis receives different scales to represent each of them. The number of metrics displayed at the same time is limited to eleven. Panning and zooming within time histograms are possible through the use of scroll bars and buttons in the graphical user interface. With that, users can navigate over time to see the evolution of each metric. The barchart visualization enables the visualization of data in real-time and it is designed to view a considerable amount of metrics. The drawback of this view is that it has no historical representation. The display has as horizontal axis the different metrics being analyzed and in the vertical axis the different components of the parallel application, for example. The third standard display of Paradyn is the Table Visualization. The view actually shows the data textually: columns are metrics and rows are parts of the application, typically source files or a specific function. The data in the table is updated in real-time. The fourth display of Paradyn is the 3D "Terrain" visualization. It allows the performance data to be analyzed using a surface rather than curves, as in the time histogram, or bars, like the barchart visualization. The three dimensions allow the visualization of two different metrics at the same time and their evolution over time.

The Paradyn visualization tool is still developed at the Paradyn Parallel Tools Project, with publications in 2008. New developments of the tool include STAT – Stack Trace Analysis Tool [4] and challenges to petascale tool development [50]. Paradyn's main contribution is the dynamic instrumentation of parallel applications. This idea was materialized through the W3 Search Model. Besides that, it is important to notice that the tool is available for at least 14 years, since its conception in 1995.

2.2.5 Vampir

Vampir [60] was initially developed at the Julich Research Center in Germany, but later on transformed in a commercially available tool managed by Pallas GmbH. The tool appears after the definition of the MPI standard, being one of the first tools to be able to visualize the behavior of MPI parallel applications over time. After its creation, Vampir development goes toward scalable analysis of parallel applications [14] and to analyze hybrid OpenMP/MPI applications [80]. Some of the visualizations provided by the tool are depicted in Figure 2.5.

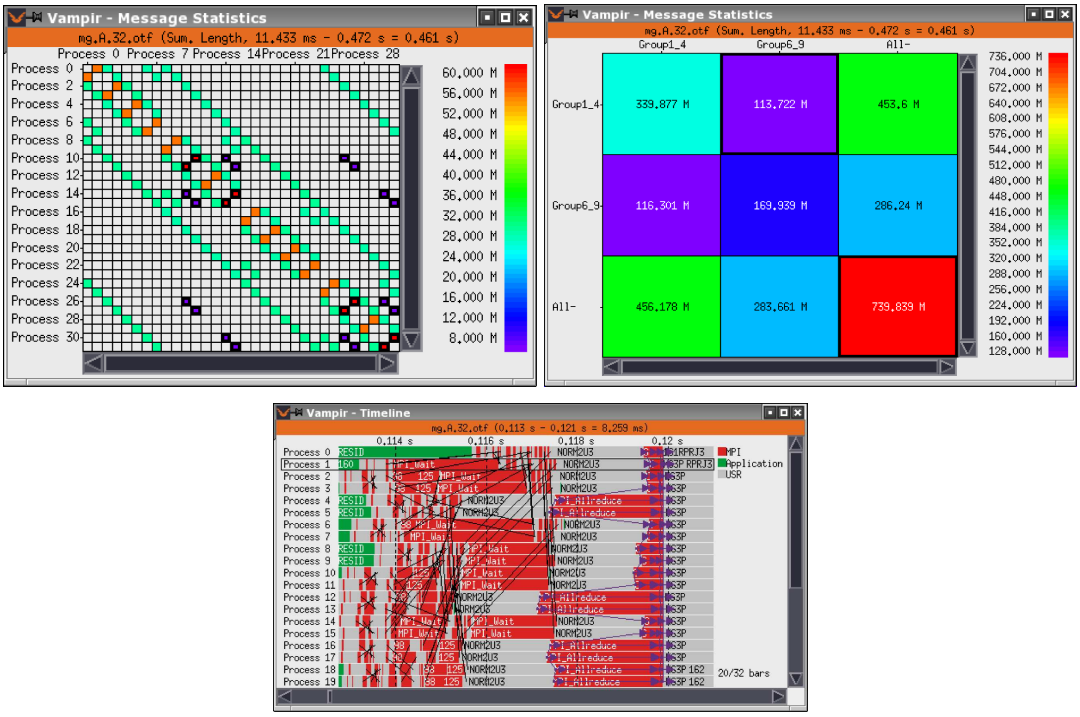


Figure 2.5 – Visualizations of Vampir, using the matrix technique (top left and top right) to summarize communications and its Gantt-chart (bottom).

Vampir has a set of flexible filter operations, which are used to reduce the amount of information displayed and to help its users to spot more easily performance problems. Another feature of Vampir is the possibility to read trace data that is distributed across many computers, in a cluster or grid-like environment.

In its efforts to turn the tool more scalable, the Vampir team developed a hierarchical visualization based on Gantt charts [14]. In this view, users navigate through data in different levels of abstraction such as cluster, machine, process and thread. The technique they propose attacks a major problem of Gantt charts, where the vertical size of the screen is a limit to the number of entities that can be analyzed at the same time. Without this technique, Vampir is able to analyze up to 200 independent objects at the same time. When applied, it allows the visualization of at least 10000 processing entities, even if only 200 are shown on the screen at the same time. The hierarchical structure of their model allows up to 3 layers. This hierarchical visualization works for timelines and statistical displays of Vampir.

The performance visualization available in Vampir can be divided in different categories: single time system snapshots, when data for a point of time is shown graphically; animation, giving the users the possibility to analyze step-by-step the dynamic behavior of the application under analysis; statistics, that are able to summarize system behavior for the interval of time under investigation; and a time-line system view, showing detailed system activities with a time axis. The visualization techniques applied include matrix chart, summary chart, Gantt-charts,

summary timeline and counter timeline.

Vampir is the tool available commercially. It uses a specific trace format and a set of programs that can be converted from other formats to the one used by the tool. Its space-time view attacks the scalability problem by proposing a data aggregation mechanism to reduce the amount of data that is visualized at one time.

2.2.6 Virtue

Virtue [72] is developed at the University of Illinois at Urbana-Champaign. The main objective of the tool is to offer an immersive visualization environment for the analysis of performance data from parallel applications. It is the first attempt to use virtual reality in the performance analysis domain. The tool connects to Autopilot [67] to receive its monitoring data and helps the performance analysis by trying to enhance rendering with human sensory capabilities.

As visualizations, Virtue offers three types of 3D visualization, depicted in Figure 2.6. The first is the wide-area geographic display, where nodes are placed following their geographic location. The second is the time-tunnel display, showing a cylinder where the internal part of the cylinder is used to represent processors state evolution over time and chords illustrate cross-process interactions. The last is the call-graph display, which forshows in a 3D space the functions that were executed and the call procedures among them.

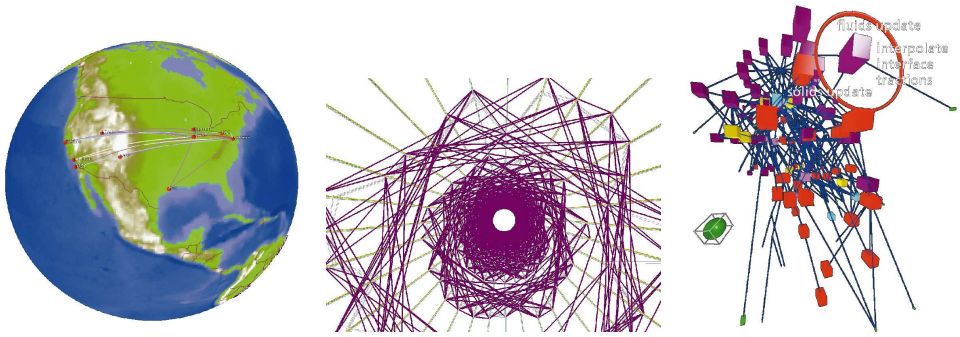


Figure 2.6 – Virtue's 3D visualizations, from left to right, the wide-area, the time-tunnel and the call-graph displays.

Although not further explored, Virtue is the first to try to use virtual reality combined with 3D graphical representations in the analysis of parallel applications. It was developed by the same team that created Pablo (see Section 2.2.3).

2.2.7 Jumpshot

Jumpshot [83] is developed at the Mathematics and Computer Science Division at the Argonne National Laboratory, in the United States. Its authors have participated in the development of the MPI specification and the release of the first draft. Currently, the development of the tool is attached to the MPICH implementation of MPI. The tool is written using Java, designed to receive a file format with time-stamped events. Initially, the file format to be used was called

CLOG. With the evolution of parallel and distributed systems, especially related to scalability issues, the file format also evolved to SLOG, and now SLOG-2 [17]. Jumpshot is now in its fourth version, providing accumulative enhancements such as previews to increase detail as needed in the timeline window.

Jumpshot offers the traditional package of visual graphs, such as histograms, accumulative state durations and series of zoomable and scrollable timelines. Two examples are available at Figure 2.7. A more specific type of visualization is called the "mountain range" view, showing the aggregate number of processes in each state at each time.

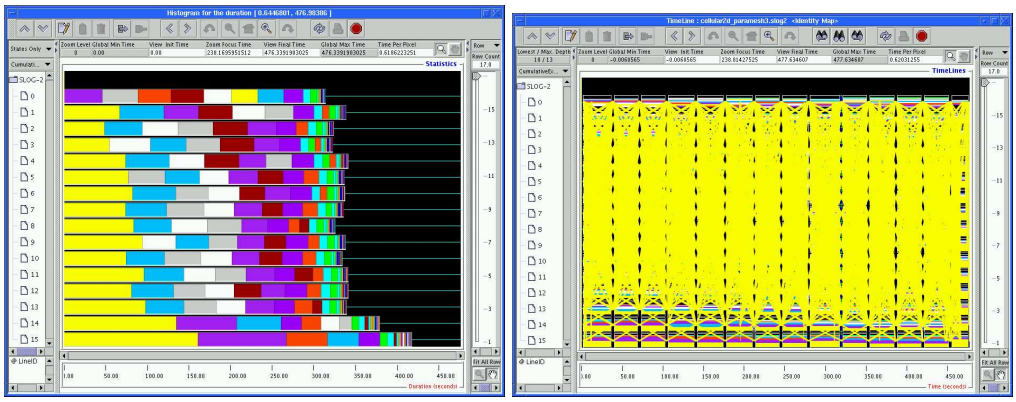


Figure 2.7 – Histogram and Gantt-chart view of Jumpshot.

Probably the most evident contribution of the Jumpshot series tools is that it is tightly coupled with a MPI implementation. This facilitates its use for MPI users, since a small period of time is needed to understand the way the tool works.

2.2.8 ParaProf

ParaProf [10] is a portable, extensible and scalable tool for parallel performance profile analysis. It attempts to unite, as its authors say, the “best of breed” capabilities already proposed in other tools. The tool was initially focused on profiling techniques, rather than using tracing techniques as other tools did. Today, the tool is able to deal with traces gathered from parallel application executions. The group that develops ParaProf has also proposed a framework for data mining [41]. ParaProf is integrated in a bigger project named TAU – Tuning and Analysis Utilities, that is being developed jointly by the University of Oregon, Los Alamos National Laboratory, in the United States, and Julich Research Center, Germany.

The architecture of ParaProf has four key components: the Data Source System (DSS), the Data Management System (DMS), the Event System (ES), and the Visualization System (VS). Well-defined interfaces are used for each component so they can interact with each other at the same time they run separately. This organization allows the tool to be extensible and flexible, enabling the evolution of the tool to other programming paradigms and new techniques.

The visualization system component of ParaProf’s architecture is responsible for creating visual representations of the data. They are based on Java2D, but 3D visualizations are also

present to represent profile data. There are four categories of visualization in the tool: 3D-visualization, thread based displays, function based displays, and phase based displays. The 3D visualizations are rendered using OpenGL hardware acceleration techniques. Each window has rotation, translation and zooming capabilities. There are three types of visualization in this category: the Triangle Mesh Plot, that shows two metrics for all functions and all threads. The height represents one metric and the color another. The resulting visualization creates a surface where data is represented; the 3D Bar Plot, that works like the mesh, but using bars; and the 3D Scatter Plot, that uses points instead of mesh or bars. The other category is the Thread Based, with a series of graphs that show statistics of the application and also a call graph, all related to the threads of the parallel application. The third category is the function based displays, composed of two views that show statistical data: a function bar chart and a function histogram. The fourth category is the Phase Based displays, focused on showing statistical data from pre-defines phases of the parallel application. Examples of the views generated by ParaProf are available in Figure 2.8.

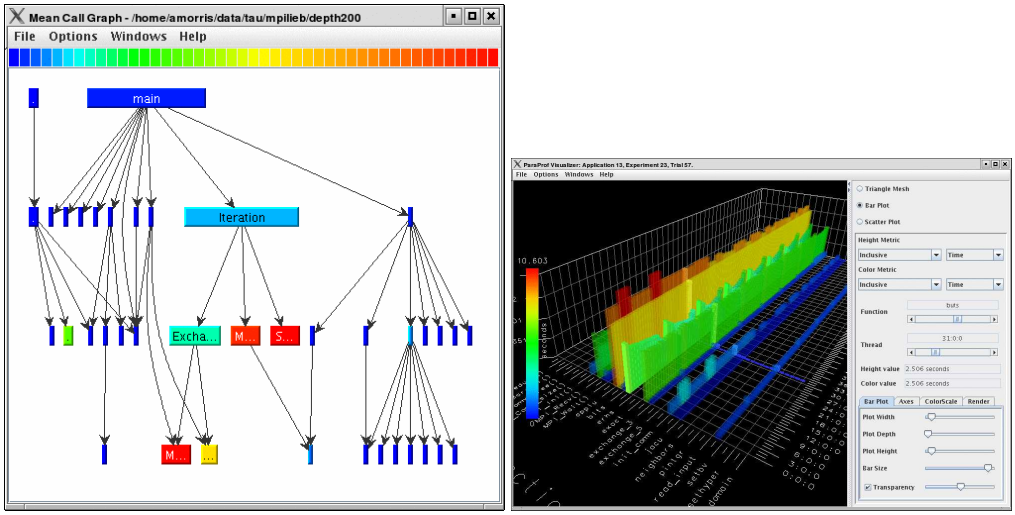


Figure 2.8 – The call-graph and the 3D bar plot of ParaProf.

ParaProf has a modern design in its software implementation, through separate components that interact with a defined programming interface. Besides that, it provides an extensive set of visualization techniques, and it is tied to the TAU project.

2.2.9 Pajé

Pajé [24, 47, 76] is a generic visualization tool designed to be interactive, scalable and extensible. The tool was initially developed at the LIG Laboratory (former ID Laboratory), in Grenoble, France, but is now developed at the Federal University of Santa Maria – UFSM, Brazil. The interactive part of Pajé means that the user is able to interrogate monitored entities, through its time-space visualization window. The scalable feature of Pajé is related to the possibility to cope with a large number of program entities, such as threads and processes, and the details

about each of them. The extensibility of the tool relates to the easy addition of new features, new types of traces, new graphical displays, new programming models to adapt the tool to the evolution of parallel programming interfaces and visualization techniques.

The Pajé file format is also part of the visualization tool. The format is textual and without semantic, where events describe the behavior of a set of monitored entities. The basic types that can be used in the format are container, state, event, link and variable. Containers can be used to group other types, creating a hierarchical definition of types. Virtually any kind of monitoring system or trace generation tool can use these types to describe the behavior of monitored entities, from parallel applications to distributed resources of a parallel system. This level of flexibility in the description of monitored entities behavior is not found on related work. If the trace file has information about source-code correlation in events, the user will be able to click-back to see which part of the source code caused the creation of a visual object rendered in the graphical displays of Pajé.

The architecture of Pajé is composed of modules that are connected through a graph that is usually fixed, but can be changed to adapt the tool to new types of components. The components can be any self-contained part that behaves following a certain protocol and operates over the events that are read from trace files. The traditional set of Pajé components includes a trace file reader, a event decoder, a simulator, a storage controller, aggregation, reduce and ordering filters, for example. Despite the number of components, the three classical components of Pajé are the controller, trace readers and the simulator.

Pajé offers to its users two types of visualization techniques to represent graphically containers, state, events, variables and links. The first and most used is the space-time window, which actually draws a Gantt-chart display improved with arrows to represent interactions among processes. The second type of display is used to dynamically show statistical information about a selected slice of time in the space-time window. These two techniques are represented in Figure 2.9.

Probably the main feature of Pajé is its flexibility. The tool was originally used to visualize traces from Athapascan applications [32], but it evolved to visualize traces obtained with Java applications [20, 62], message-passing parallel applications, thread-based applications and hybrid approaches. It was also used to see related monitoring information with a multi-level approach [70] as, for example, traces from application-level (MPI) and traces from resources and operating systems. Pajé's simulation component, the core of the tool implementation, and the aggregation filter, are able to handle a big amount of trace data spread in long periods of time.

2.3 Summary of Visualization Techniques

The last two sections addressed the historical evolution of the performance visualization area and the description of a representative set of visualization tools for parallel applications. The objective of this Section is to try to summarize the visualization techniques used. We divide the techniques in three types: behavioral, structural and statistical. When possible, we make reference to the tools that implement these techniques.

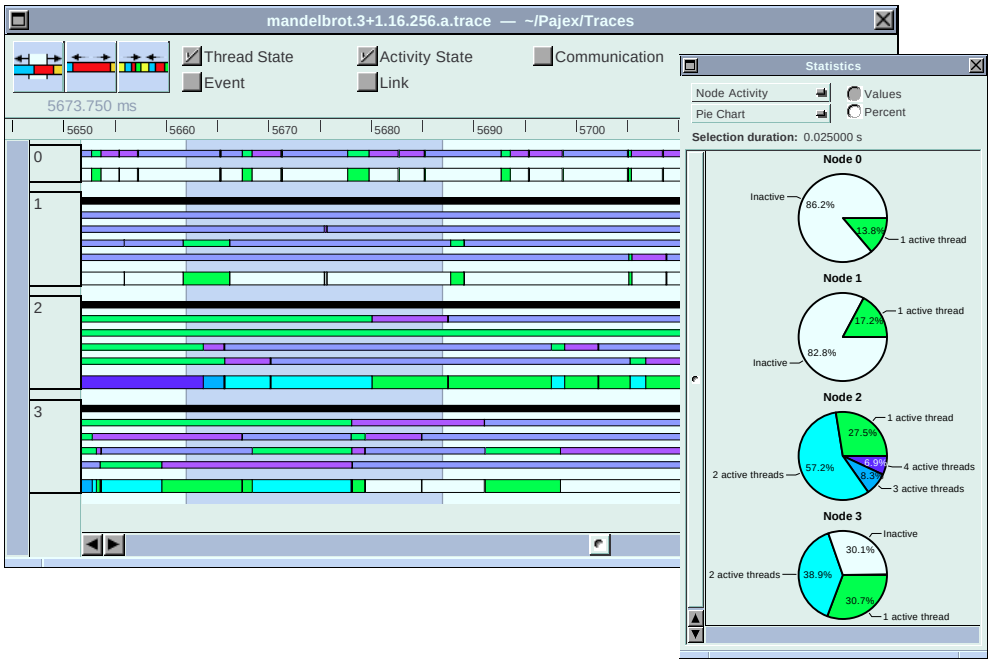


Figure 2.9 – The two visualizations of Pajé, including its space-time view and the pie-chart statistical view.

2.3.1 Behavioral

This Section presents the visualization techniques that have a timeline and show the behavior evolution of metrics and components through time.

Gantt-Charts

Gantt-chart [79] is a visualization technique created more than 100 years ago. Initially, it was used to organize and schedule the tasks of projects. It was one of the first techniques to be used to analyze parallel applications. Figure 2.10 shows a simple Gantt-chart with the behavior evolution of a set of entities. These entities can be anything related to the parallel application or the execution environment. For each of them, the rectangles represent a state that has a duration in the timeline. Arrows can be used to illustrate an interaction between two entities. This type of visualization can also be used to show the user the critical path of the parallel application. ParaGraph, for instance, has a special feature about that.

Almost all tools that show performance visualization implement a Gantt-chart like technique. In some of them, this type of representation is called “Spacetime”. Although very useful to represent the behavior of a set of processes from a parallel application, the common issue with Gantt-charts is related to scalability. Computer screens are limited in terms of vertical resolution, and this is reflected in the technique. Some tools such as Pajé and Vampir implement hierarchical grouping mechanisms that allow the observation of a larger number of processes.

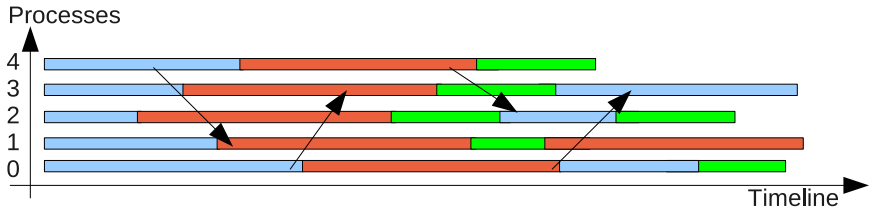


Figure 2.10 – A simple Gantt-Chart showing the behavior evolution of an application with 5 processes: the bars indicate different states and arrows indicate interactions between processes.

Variables in two and three dimensions

This type of display is a graph where one of the axis represents time. Figure 2.11 shows an example with two metrics being displayed. The vertical axis represents the values that the variable can reach over the period of time being analyzed. Observing a significant metric can give hints about the CPU or memory utilization of a machine during the execution of a parallel application. Almost all performance visualization tools also provide some sort of representation of variables behavior through time. Examples are the “variables” visual object of Pajé, the “Communication Traffic” and the “Utilization Count” displays of ParaGraph, and the “Performance Counters” representation of Vampir.

Two cases that are similar to this 2D approach is the “Time Histogram” of Paradyn, where performance data for metric/focus pairs are represent with a time axis (focus is a piece of code of a parallel application); and the “Node Statistics” technique of ParaGraph, when a specific metric is shown for one node with a timeline.

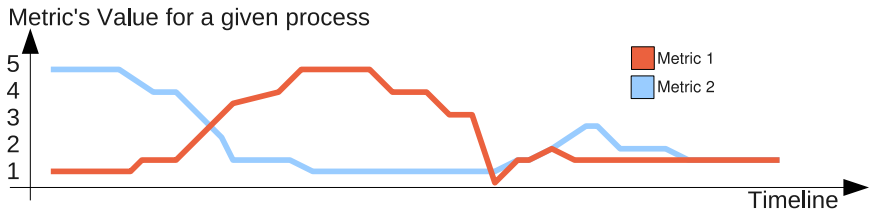


Figure 2.11 – Showing the evolution of two different metrics over time.

Another visualization technique extends the 2D approach by combining two related metrics and representing them with a timeline. This 3D approach can in fact show more information to the users. The technique is named as “3D Terrain Visualization” and is present in Paradyn.

Time-Tunnel

The only occurrence of the time-tunnel display is on Virtue. The technique works under 3 dimensions, where two of them are used to place processes, in a circle, and the third dimension represents time. The observation point is placed in the middle of the circle. The interactions among the processes are placed within this 3D environment, taking into account the position of

processes and the time of occurrence. The resulting visualization looks like a cylinder, where the user observes arrows crossing the interior of the cylinder. Figure 2.6, of previous Section, illustrates the approach.

Phase Portraits

Phase portraits are the result of a technique commonly used in other areas of science, such as physics. They show the evolution in time of two related variables, or metrics. Figure 2.12 shows the resulting visualization. The performance data is collected through a period of time, between regular intervals. The idea is to create points in the graphical representation and connect these points following the order in time among them. ParaGraph is again the only tool to implement this technique.

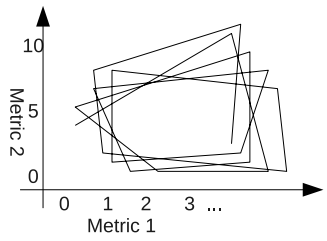


Figure 2.12 – A phase portrait showing the relation among two metrics.

2.3.2 Structural

This Section presents the visualization techniques that try to visually represent the structure of applications. By structure, we mean the different types of relations that connect the components of parallel applications, such as processes and threads.

Call Graphs

Call graphs are used to give to the user a representation where the interactions among the application's components are shown. Figure 2.13 is an example of that. Nodes can represent functions or methods, and the arrows between them represent a function call or method call. This method of visualization is especially useful in the analysis of parallel applications that are organized as a data-flow graph.

Some tools implement this technique, such as ParaProf and Virtue. The latter implements call graph within a 3D environment, giving the user different forms of interaction to highlight parts of the graph with additional information, such as the name of the node, associated values and so on. This was implemented to avoid the representation of all data for large graphs.

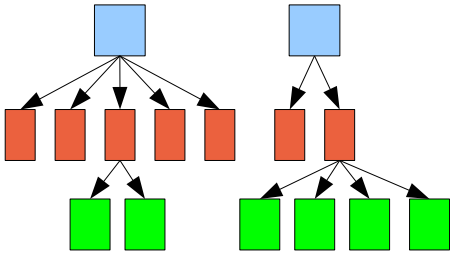


Figure 2.13 – The call graph displays showing the function call of two regions of a given program.

Matrix

The matrix of communication is a technique where a two dimensional representation is organized with one of the axis showing the senders processes, and the other axis, the receivers. For a point in time, the matrix shows different pairs sender/receiver by coloring the matrix. Colors can also be used to show additional information, such as the type of the communication, if it is collective or not, or the size of the data transmitted. The left image of Figure 2.14 depicts this technique.

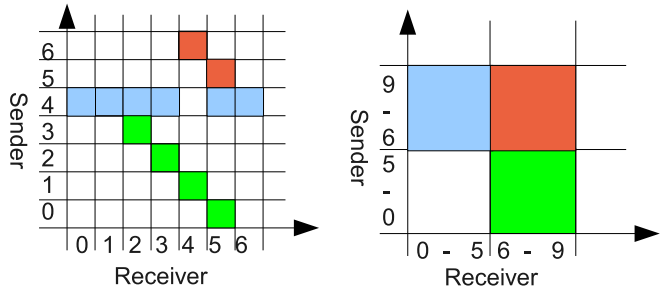


Figure 2.14 – Matrix of communications among processes and also the grouping technique.

ParaGraph was the first to propose this technique, with a limited number of processes involved. The scalability of this approach is related directly with the number of processes. Vampir tries to solve this problem by grouping processes according to their number of other characteristics. This is shown in the right image of Figure 2.14.

Graph with Communications

A graph is used in the ParaGraph tool to represent the communications among a set of processes in a given time. The Figure 2.15 illustrates the approach, with the communication pattern among three processes. ParaGraph has also a set of pre-defined hardware interconnections, such as the Hypercube, and allows the observation of which links are used by the application at a specific point in time. Different layouts for the hypercube representation were possible, such as the linear view. The problem of the approach of ParaGraph is that no additional information about the

links were provided to the user. The technique was used only to show when a certain interaction happened during the application execution.

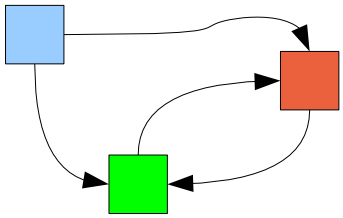


Figure 2.15 – Communication pattern with three processes for a given time.

2.3.3 Statistical

This Section presents the visualization techniques used to represent statistical data based on the traces available for the analysis.

Bar and Pie Charts

Bar and Pie charts are a traditional way to show the values of a certain metric for a number of processes. For example, they can be used to show how many messages a process has received, or the amount of memory used in a machine. Figure 2.16 shows an example of a barchart and another example of a piechart.

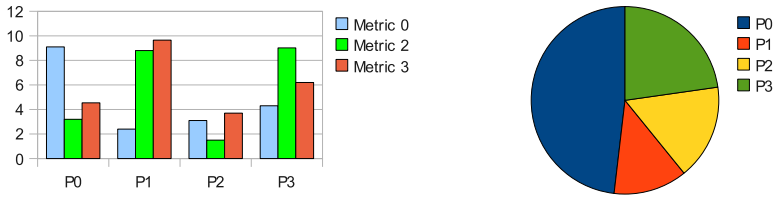


Figure 2.16 – Barchart and Piechart displays.

These types of charts have been available since ParaGraph. Other tools have also implemented them using different metrics and techniques. Paradyn, for example, implements horizontal barcharts with more than one metric, each of them with one different scale in the horizontal axis. Pajé's piechart implements the technique to quantify, in a given period of time, how much time a certain process spent in different states. The user can then compare two processes to look for performance problems.

Kiviat Diagrams

Kiviat diagrams, also known as radar map, are a chart that consists of a sequence of equally distributed spokes, each one representing one of the monitored entities. In the area of performance visualization, the spokes are used to represent processes, and each process has a scale

of value for its spoke. Then, for a given metric about one process, a point is chosen in the spoke. Connecting these points form a geometric figure that can be used to detect irregularities among processes, if a similar value is expected for all of them (load balancing, for example). Figure 2.17 shows a schematic example of the technique, with 3 metrics shown for 4 processes.

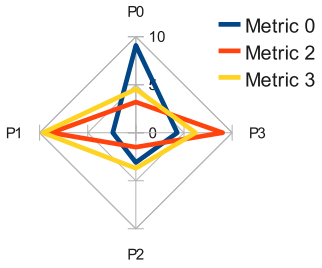


Figure 2.17 – The Kiviat Diagram for 4 processes with 3 different metrics.

ParaGraph has been the only tool to implement the technique. This type of display also has scalability issues when the number of processes or related metrics increases. After its first appearance in ParaGraph, no further development to solve this scalability issue has been present in other tools.

Statistical 3D representations

3D representations without a time axis are already present in the literature. The idea is to plot in a tri-dimensional space drawings that are generated using three different metrics. The ParaProf tool has three displays that follow this design: “Triangle Mesh Plot”, “Triangle Bar Plot” and the “Triangle Scatter Plot”. The first connects the points using a mesh, resulting in a visualization like a terrain with elevations in some points. The second represents data as vertical bars and the last just draw the points in the 3D space.

2.4 Summary

Several visualization techniques exist today for the analysis of parallel applications. These techniques help the developer to obtain a better performance and also provide a way to understand the behavior of programs in a given execution environment. A possible classification of the visualization techniques is the division in three types: **behavior**, such as the space/time and phase-portrait views, showing the evolution of entities over time; **structural**, focused in the observation of communications, such as the techniques matrix, communication graph and call graph; and finally **statistical**, which summarizes trace data.

The two next Chapters present the visualization techniques proposed in this thesis. In the beginning of each Chapter, we show that existing visualization tools are not fully suitable for the analysis of grid parallel applications. The first Chapter deals with the lack of support from visualization tools to the analysis of parallel applications mixed with network topology. The

second Chapter proposes a visualization scheme that achieves visualization scalability and can be used to analyze parallel applications composed by thousands of processes.

Chapter 3

The Three-dimensional Model

The previous Chapter has listed tools and techniques that can be used to analyze the behavior of parallel applications. The presented tools were detailed in terms of features and capabilities, including which visualization techniques are implemented. At the end of the Chapter, we presented a classification of the techniques in three types: structural, behavioral and statistical. Generally, most tools were built to handle precise environments, such as clusters, where the dynamics of the resources are not felt by applications since usually the access to the resources is made exclusively. This Chapter goes through the grid characteristics to show that the traditional visualization schemes are not able to fully help the developer to analyze parallel applications, particularly when network characteristics must be taken into account.

The performance of grid parallel applications is directly related to the characteristics of the network interconnection [49]. When the grid resources have a strong hierarchy among them, as in the case of a lightweight grid, the choice of resources given to an application can be decisive for its performance and later understanding of its behavior. For instance, if two sets of processes perform more communications between them and are placed in two distinct locations of a grid that does not offer the lowest latency, the application can suffer a loss in performance. Sometimes, the analyst is not able to make the link between application and network characteristics. The decisions taken from a traditional analysis may lead to wrong conclusions about the bad performance. In this case, if we were able to analyze the application behavior together with the network characteristics, we would see more clearly the reason of the application behavior.

This example can be more explicit if we consider that each parallel application has a communication pattern. These patterns are defined when the application is implemented, through the use of paradigms such as master-slave, divide-and-conquer and so on. During an application analysis, it would be interesting to visualize this pattern together with the network topology. With this, it would be possible to optimize the match between the network interconnection and the application's communications. If this optimization is not possible, the analysis could be used to help the developer to adapt the application in order to better explore the network characteristics.

Looking at the tools presented in last Chapter, we can notice that most of the techniques they present are not able to handle an analysis that takes into account the network interconnection. ParaGraph (see Section 2.2.1) is the only tool that has the notion of interconnection

in its visualization techniques, although providing only hypercube visualizations and program communication patterns, separately. In fact, ParaGraph was not designed to analyze large-scale applications, with thousands of processing entities. Other techniques, such as the space-time visualizations or graph-based views, present in almost all visualization tools, are also not able to depict the network interconnection together with the communications of parallel applications. In this case, the limitation is related to the way resources and components of application are drawn, which is made on a linear space. As the architecture gets larger and more complex, highlighting its topology becomes impractical. And even if some sort of simple topology organization can be represented using one of the axis, labeling the platform representation with additional characteristics like throughput and latency usually degrades the readability of the whole picture.

Our proposition to make a link between application analysis and network topology is based on a visualization scheme composed of three dimensions. One of the dimensions is the timeline, where the components of application can be analyzed using a behavioral view. The other two dimensions are used to draw either a structural or statistical representation. In the context of the problem being addressed, these two dimensions are used to draw a visual representation of the network topology. Broadly speaking, our proposal combines at different levels the three types of visualization techniques we discussed in Section 2.3, resulting in a mixed behavioral-structural/statistical representation.

Some visualization tools for parallel application analysis already have 3D visualizations. ParaGraph, for instance, has a 3D representation for a Torus Network Topology, but its focus is in the instantaneous analysis of the interconnection utilization, with no axis reserved to work as timeline. Another example is Paradyn, that contains its 3D Terrain Visualization being able to show the relation between two metrics and their evolutions over time. Since the two dimensions of the 3D Terrain are not conceived to draw graphs, Paradyn is not able to visualize the network topology and application evolution at the same time. The third example of tool that uses 3D visualizations is Virtue. Among its visualization techniques, the time-tunnel is the only one that seems like our approach, but it is fundamentally different, since it was not developed to show the network topology or parallel application communications pattern. Virtue only places the processes of an application in a circular manner in two of the dimensions, letting the third dimension act as timeline. The view of the developer is always pointing to the center of the circle. Communications and interactions are drawn inside that circle, in a 3D space. TAU's ParaProf also has its 3D visualization, but focused on the analysis of statistical data. This means that ParaProf is able to visualize three types of related events in the same visualization, using the three dimensions. However, ParaProf is not able to use one of these dimensions as timeline and it is incapable of drawing graphs in the two remaining dimensions. In summary, we can see that there are tools that already provide some sort of 3D visualization, but none of them have the same approach as we have, merging network topology to the application analysis.

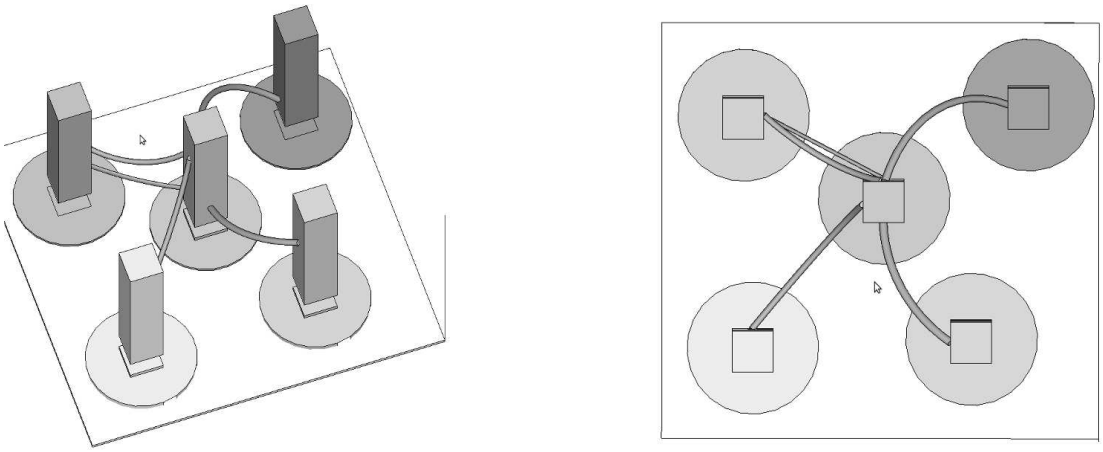
The rest of the Chapter is organized as follows. We start by describing the visual conception of the 3D approach, detailing its visual objects and how application traces are mapped into the 3D view. In Section 3.2, we explain the abstract model that deals with the monitoring data and generates the 3D visualizations, followed by a series of sections, each one describing the components of the model: the trace reader, the extractor, the entity matcher and the visualization component. During the description of the entity matcher, we detail three configurations that can

be used inside the 3D approach.

3.1 Visual Conception

The visual conception of our model consists in the combination of visualization techniques that show the behavior of the application with techniques that show the structure or statistical data. If a structural data is used in combination with the behavior representation technique, the user can observe the evolution of monitored component through time and consider the structural organization. This is the case when users have to analyze the parallel application with the network topology, for instance. If statistical data is used instead, the user can summarize in quantitative terms the behavior of the application, using different time scales and slices. In a more practical way, these combinations allow the representation of the notion of gantt-charts combined with graphs and summaries.

The result of this visual conception is the three-dimensional model. The model has two dimensions reserved for the representation of a structural or statistical view. We named these two dimensions the visualization base of the 3D model. The third dimension is the timeline. Figure 3.1(a) shows an example of the 3D approach to represent application data. The states of the processes are represented in the 3D visualization as vertical bars. They are placed on top of the visualization base. The different states along the time axis of a certain process are represented by different colors. Each state representation is placed vertically following the start and end timestamps. Communications can be represented as arrows or links within the 3D environment, connecting two or more processes that communicate. The Figure 3.1(b) shows a different point of view, located on top of the visual objects. This vision allows the observation of the communication pattern of the application.



(a) Visual conception of the 3D approach. (b) Top-view of the same scene.

Figure 3.1 – The visual conception of the 3D approach with application traces represented by vertical bars showing processes behavior through time.

The visualization base of the model is composed of two dimensions. They are used to depict

either structural or statistical representation techniques. Structural representations, as presented in Section 2.3, can be mainly graphs and matrices or any other technique used to organize the components of the application. Statistical techniques can be used to summarize a particular part of the behavior of the components being visualized.

Lots of configurations are possible for the visualization base. For instance, it can be used to illustrate the communication pattern of the parallel application, but also the network topology involved in the execution of a parallel application. In our model, we propose three types of configurations for the visualization base (see Section 3.5). Two of them are structure-based, showing interconnection graphs. The other is an information visualization technique called Treemap [42], used to represent hierarchical information data. Additional techniques can be easily adapted to our model to work as the visualization base.

The third dimension of our model is the timeline. It is usually represented as the vertical axis of the 3D approach, as can be noticed in Figure 3.1(a). The timeline axis is used to show the component's behavior evolution through time. In the case where the components are processes, the vertical bars that represent them might have different colors to represent states and arrows to represent point-to-point or collective communications. These representations characteristics are similar to the ones present in space-time views, but here in three dimensions. The timeline is configurable to offer the users different time scales that can be dynamically changed.

When using graphical visualizations, users are interested in interaction mechanisms, like zooming, online information updates and so on. They improve the user perception of specific parts of the information, enabling a deep application and platform behavior analysis. Animations can also be applied to dynamically change the graphical visualization. Resizing rectangles and changing their colors to reflect the platform state in given time intervals are some examples. In this case, changes are caused by continuous information updates coming from the monitoring system. Another type of graphical interaction mechanism is constituted by distortion techniques [15], which magnify only specific parts of the representation. The fish-eye technique [69] is a good example of such technique. It helps the user to obtain details about a picture area without losing its context (as opposed to a simple zoom).

Besides these interaction mechanisms, we have a set of possible interactions with the 3D approach. An example of that is the notion of observation point. In this context, the view that the user is staring at any time is generated by a camera. This camera can be moved inside the 3D space with rotation, translation and approximation techniques. This allows multiple views of the same data, from different angles.

3.2 Model Overview

In order to create a 3D visualization, the trace data collected from the application execution must pass through a series of transformations. We define here an abstract component model, in which these transformations are detailed. Figure 3.2 depicts the overall organization of the model. As input, the model uses two types of information: the trace files from the monitored application and a configuration file that holds the resource description of the execution environment used by the parallel application.

The visualization base is configured by the entity matcher module (C). We have implemented

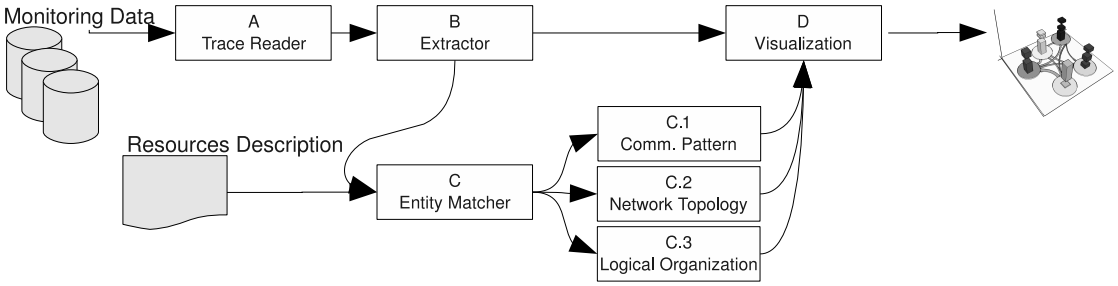


Figure 3.2 – Abstract Component Model of the 3D approach, with the three different configurations for the visualization base (represented by C.1, C.2, and C.3).

three different configurations for it (they are detailed in Section 3.5): one that shows the communication pattern of the application; another that shows this pattern combined with the network topology of the execution environment and the last one is the combination of application data with a logical organization of the resources. The entity matcher chooses one of these configurations based on the resource description defined by the user.

Among the three alternatives modeled in the Entity Matcher, the one that considers the network topology (C.2) directly addresses the problem regarding the influence of the network interconnection in the application. The additional two alternatives are presented to show other structural information (the communication pattern) and statistical representation together with behavior details through time.

We consider in the model that the trace data is available as trace files, under the form of a flow of events that traverses the components of the Figure 3.2 from left to right. Nevertheless, even if we take trace files as input, the components are described independently of how trace data is offered to the model. Therefore, the model is able to deal with an online generation of events in case the flow of these events is not so bandwidth intensive. Notifications can also occur from the visualization component to the others, in a right-to-left fashion, in order to propagate configurations and behavior changes triggered by user commands.

Next sections detail the components of Figure 3.2. We start by explaining the Trace Reader (A), including the mapping from the trace events to the objects used by the model. Section 3.4 shows the Extractor (B), followed by the description of the Entity Matcher (C), considered as the main component of the model. We end the description of the model with the Visualization (D) component.

3.3 The Trace Reader

The generation of traces during runtime is a classical technique to record the behavior of parallel applications. If applied carefully with large memory buffers and a selected set of events, it can be used without disturbing too much the natural application behavior. In large-scale parallel applications, it is common to generate one trace file per process. After the end of the application execution, the different files are gathered and merged with different transformation techniques. This is modeled by DIMVisual [70], which is a data integration model for visu-

alization of parallel applications. The model uses the synchronization technique developed by Maillet and Tron [52].

One trace file is usually composed of events. An event has a type, a timestamp and additional information that goes with its type. They can be used to trace a high number of information in parallel applications. The classical points where trace events are generated are the beginning and the end of both communication and processing functions. Point-to-point and collective are commonly traced with events, registering the exact point in time that a message is sent and received. Although most of tracing mechanisms generate timestamped events, this association with time is not a requirement. Events can be used to simply count the number of times a certain behavior occurs, for example, without the need to know when it happened. Another characteristic of the events of one trace file is that their timestamps might not be synchronized with the events from other files. This happens because they are generated in different machines, with different clocks.

The trace reader component is the only part of our model that deals directly with application traces and events. Its responsibility consists in reading, synchronizing and transforming them into high-level visual objects. Although these objects represent the content of traces, they have no semantic data and can be managed in a generic way. This allows the rest of our model to be independent from the trace file format. The high-level representations are mainly composed of entities, states and links. An entity can be a process, a thread, or a machine. Generally speaking, an entity can be anything that is observed during a period of time and is related to the application analysis. States and links are always related to one or more entities. A state is defined as the behavior a certain entity may have during a period of time. A link is used to represent an interaction among two or more entities in a time interval.

Figure 3.3 shows the behavior of the component. The trace data is represented in the left of the Figure, showing the events that are in different trace files. In this example, we list the beginning of the behavior of two processes, through 8 events already ordered. Process 1 starts, then sends a message; and process 2 starts and blocks to receive the message from 1. The trace reader transforms these events into the visual objects depicted at the right of the Figure. In the example, they were transformed into two entities, P1 and P2, to represent processes; two states, Send (created with *send_start* and *send_end*) and Receive (based on *receive_start* and *receive_end*); and one link, represented by the arrow based on *msg_send* and *msg_receive*. The flow of visual objects in the output is ordered by the object's end time.

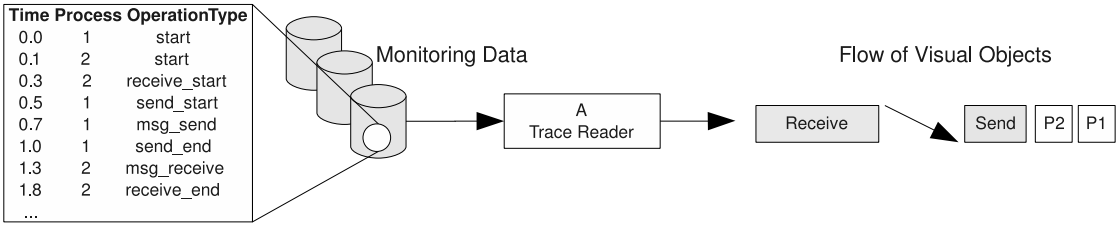


Figure 3.3 – The Trace Reader component transforms trace files, on the left, to a visual objects representation, on the right.

This component makes the rest of the model independent from the input file format. In the

case a new format is available as input, only this component should be changed or replaced, the rest of the model will continue to work in the same way as long as the output generated by the trace reader is composed of the generic entities we explained above. The trace reader sends the output to the extractor module, which is detailed in the next Section.

3.4 The Extractor

The main purpose of the extractor component is to select, from the flow of visual objects sent by the trace reader, the objects that the entity matcher component needs to work. The entity matcher is focused on the set of entities and the interactions among these entities. This means in a more practical way that it wants to know about the processes, threads and other execution flows that should be analyzed and the message exchanges, remote procedure calls and notifications that happen among them.

Taking the entity matcher's needs into account, the extractor works by observing the flow of visual objects and by selecting entities and links. Figure 3.4 depicts the behavior of the extractor with its two outputs, on the right of the Figure, considering as input the data that came from the trace reader, at left. The first output of the component, composed of the flow of visual objects received by the TraceReader, is sent to the visualization component. This enables the visualization component to be able to take into account all the data that should be used to create a visual 3D representation. The selected visual objects are sent to the Entity Matcher component, composed of the entities and links that are encountered in the flow of the input. There are 10 processes, from $P0$ to $P9$, in the example of the Figure. We have as input a flow of events with three communications ($P8 \rightarrow P5, P0 \rightarrow P3, P4 \rightarrow P2$), and six states, three send (processes $P0$, $P4$ and $P8$) and three receive (processes $P2$, $P5$ and $P3$). The output to the entity matcher is composed by the links and the processes entities, without the states.

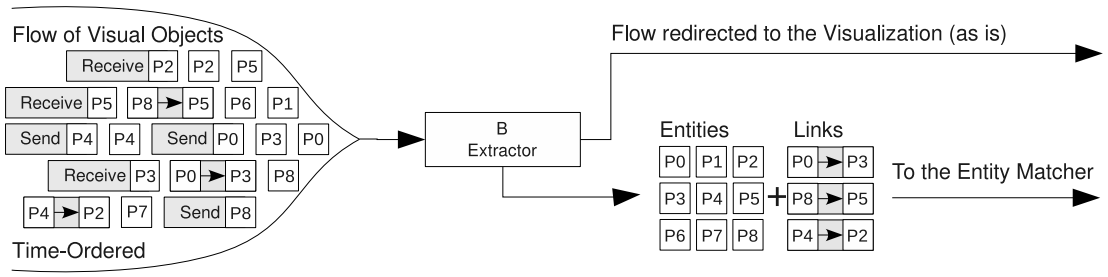


Figure 3.4 – The Extractor component selects from the flow of visual objects the entities and the links among them.

The extractor processes events and works whenever new data is available in its input. A different configurable behavior is also possible: instead of acting on a per-input basis, the extractor works on a given time interval. When this happens, the component acts by treating only the events that are present in the given time interval. This increases analysis possibilities by user interactions technique, such as zoom for a given time interval with increased details on trace data. This also influences the behavior of the entity matcher, giving the model more control in

terms of which part of the execution period will be analyzed by the user.

The extractor component is also responsible for attributing the entities with the locations where they were executed. In some cases, the entity matcher component needs this type of information for each entity. The information is necessary, for instance, when the visualization base of the 3D approach is configured to show the network topology. On this occasion, the information of where processes executed is important to correctly place them in the visualization base. For the cases where location attributes are necessary, the extractor must find such information somehow. Usually, the extractor obtains this information from the trace reader, through a specific event of the trace file format. If this location data is unavailable in the flow of objects and their attributes, though, additional input should be used, probably in the form of a configuration file.

3.5 The Entity Matcher

The entity matcher component is in charge of the visualization base configuration. It does that by taking as input the resource description set by the user and the selected visual objects with application data. The resource description is given to the component in one of two possible formats: either as a hierarchical structure describing the logical organization of the computational system, or as a graph describing the network topology of the execution environment. With the application traces and these resources descriptions, we have developed three possible configurations for the visualization base. Figure 3.5 depicts the overall organization of the entity matcher and its sub-components that implement the three different cases that are later represented in the visualization base.

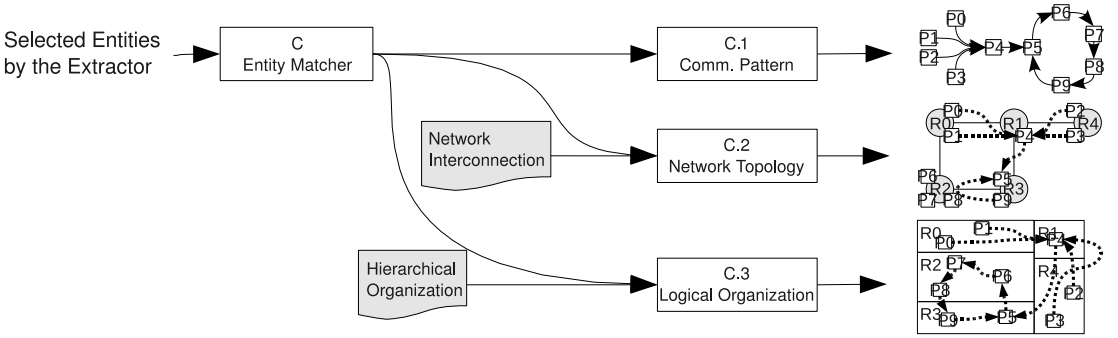


Figure 3.5 – The Entity Matcher component send its input to one of the visualization base configurations, depending on user actions.

An important aspect of the entity matcher is its extensibility. Although we have developed three different modules that illustrate the possibilities of the approach, the entity matcher could be extended to other types of organizations based on the entities and the communications representation. An example of that could be a statistical module that could group the entities according to some specification. Other types of visual representations could also be supported by the module, such as Cushion Treemaps [78] and Voronoi Treemaps [9].

The three cases we detail in the next sub-sections cover two types of visualizations for parallel program analysis (as defined in Chapter 2): structural representations, as in the cases 1 and 2; and statistical representations, as in the case 3. With these cases, we are able to combine a behavioral representation (with the timeline), and a structural/statistical representation, increasing the possibles analysis offered to the users.

3.5.1 Case 1: Parallel Application's Communication Pattern

The first configuration for the visualization base of the 3D approach shows the communication pattern of the application. The extractor component (see Figure 3.4), selects from the flow of visual objects the monitored entities and the communications among them. This selection is represented in the left most part of the Figure 3.6. The entity matcher acts by merging this information into a graph that represents the communication pattern for the selected objects. The graph creation is dynamic and based solely on the arrival of new monitoring data through the flow of events. This graph can highlight particular performance issues of the application, like bottlenecks or unbalance. Besides, it can help the developer to develop its application which uses a particular communication pattern, such as master/slave or divide-and-conquer models. Another advantage is that the application developer can see if some part of the application is overloaded with too many communications in a small period of time, increasing bottlenecks effects. The graph is then sent to the visualization component, which draws the graph in the visualization base and the evolution of the application's components in the vertical axis of the 3D environment.

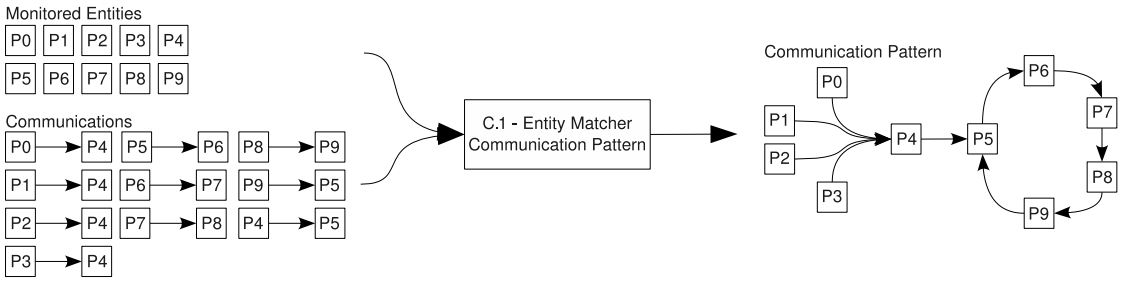


Figure 3.6 – Entity matcher configured to generate the communication pattern of the application, based on the processes and the communications.

The example of Figure 3.6 illustrates the generation of the communication pattern. The component has as input 10 processes, from P0 to P9, and a set of communications among them. As output, we can see a ring-like communication, among the processes from P5 to P9, an all-to-one communication among processes from P0 to P4 and a one-to-one communication between P4 and P5. This communication pattern can change dynamically depending on the which visual objects are selected by the Extractor module and sent to the Entity Matcher component. As previously discussed, the communication pattern can reflect the application for a given time interval.

3.5.2 Case 2: Network Topology combined with Communication Pattern

The second case for the visualization base is the combination of the network topology and the communication pattern of the application. Figure 3.7 depicts this situation, where the entity matcher receives as input the network topology (bottom part of the Figure) and the application data selected by the extractor. The application processes must have location information that defines where they were executed. This information comes with the visual objects selected by the extractor. This is necessary because the matcher needs to combine them with the resource description. As output, the component generates two graphs: one that represents the network topology itself, and another that is rendered on top of the first, showing the communications among the processes for the selected objects.

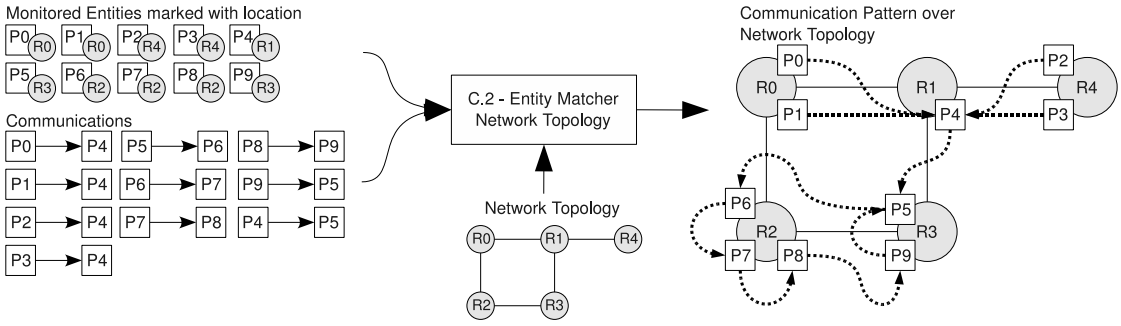


Figure 3.7 – Entity matcher can receive the network topology as resource description, creating as output the communication pattern over the network interconnection.

Figure 3.7 shows the same example as Figure 3.6, but with the network topology description as an additional input for the entity matcher. Each process has a resource associated with it, from R0 to R4. The network topology connecting the resources is on the bottom part of the Figure. The right part of the Figure shows a visual representation of the output, composed of network topology representation, with straight lines representing the interconnections, and processes on top of the resources they used during the execution. Communications among processes are represented by the arrows with dashed lines. This output is sent to the visualization module to be rendered in the visualization base of the 3D scene. The position of the processes in the visualization base will then be used by the visualization module to render timestamped events in the vertical axis. Through this combination, we are able to understand the application behavior taking into account the network interconnection of the execution system.

The developer can benefit from this configuration in the visualization by seeing the match between the communication pattern of the application and a specific network interconnection. With this match, the application can benefit more from the network, avoid concurrent communications and improving the number of parallel communications that can happen at the same time. Moreover, if the network topology has bandwidth and latency information, the developer is able, with our approach, to adapt the application in a way it obtains the highest bandwidth for the processes that communicate more data and the smallest latency for the processes that exchange messages more intensively.

3.5.3 Case 3: Logical Organization and the Communication Pattern

The third configuration is a combination of the communication pattern of the application and a logical organization of the resources. The input to the sub-component of the entity matcher in this case is the same as case 2. But for the resources, we use a hierarchical description instead of using a graph. Figure 3.8 shows the same previous example, but having as input a hierarchical structure where the resources are grouped by their location. In the Figure, the resources R0 to R4 have been grouped according to a hypothetical organization by clusters C0 and C1 and then by grid. This structure can be customized in the model to represent other types of organization, such as administrative domains or middleware dependent structures.

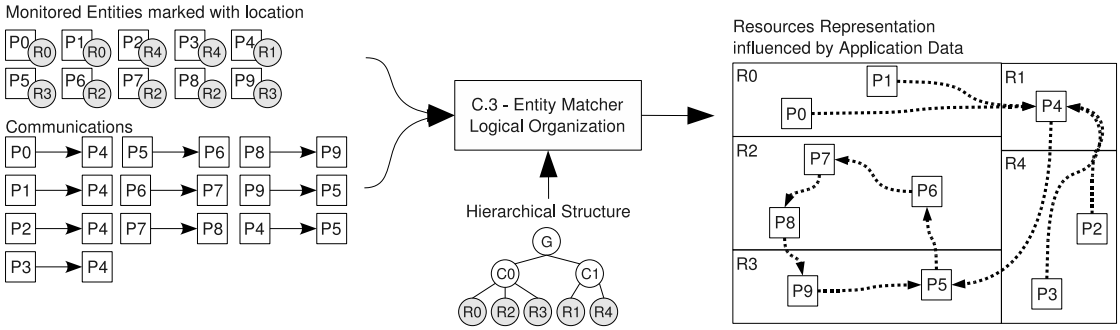


Figure 3.8 – Entity matcher configured with a hierarchical structure of the resources, generating as output a squarified treemap customized with application components.

There are many ways to visually represent a hierarchical organization. In this work, we have used the treemap concept [42] to represent them. This technique works by using recursively nested rectangles to represent tree-structured data. On the right of Figure 3.8, we show an example of treemap created using the hypothetical hierarchical structure given to the entity matcher module. Each rectangle represents a resource and its size is directly related to the amount of processes it contains. The dashed arrows are the communications rendered in the space-time part of the 3D space and reflect the communication pattern of the application. This output is sent to the visualization component, which is responsible for drawing in the visualization base of the 3D scene the treemap created by the entity matcher. An important characteristic of this configuration is that the entity matcher can be adapted to configure the treemap using other characteristics of the application data, such as the number of communications, the time spent by the monitored application executing a certain function, and so on.

The visualizations obtained with this technique in the visualization base can highlight important parts of the application in contrast with the resources. For example, it can be used to see resources usage and the load balancing of the application by configuring the treemap to show the time spent in the functions that do the processing part of the application. The same situation can be applied in order to observe which processes communicate more or stay blocked more time due to message-passing.

3.6 The Visualization

The main goal of the visualization component is to create the 3D visual representation. It does that based on the flow of time-ordered visual objects and the base configuration chosen by the user. As previously explained, the flow of visual objects is composed of entities, states and links. Since there are three different configurations for the base, the visualization component can create three different 3D representations. Figure 3.9 illustrates the component behavior, where the base configurations are at bottom, the visual objects at left and the three different visualizations on the right.

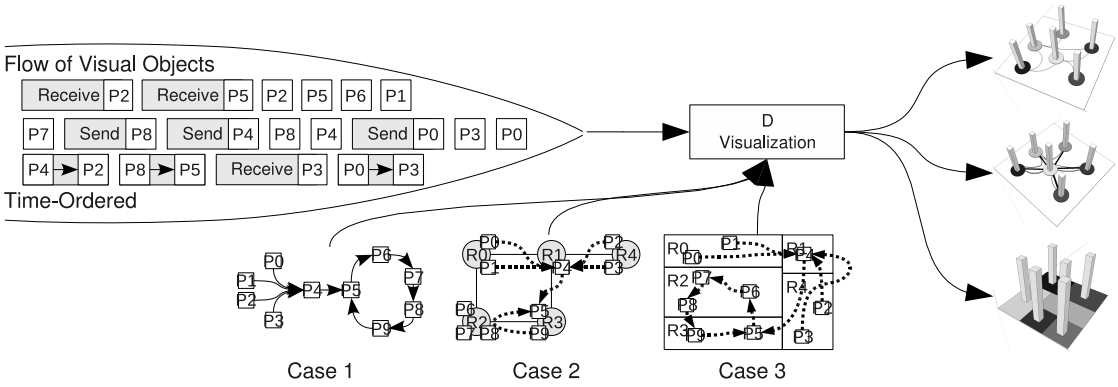


Figure 3.9 – The Visualization component receives the flow of visual objects and one of the configurations from the entity matcher, creating a 3D scene.

The timeline composes one of the characteristics of the 3D scene. It is usually rendered in the scene as a vertical line with labeled tics. The initial timestamp, usually 0, means the beginning of the application traces. It is placed right on top of the visualization base. Although this is the normal behavior for rendering the timeline, an offset can be applied if the user is interested in other parts in time of the application traces. In this case, the labeled tic that is placed just on top of the visualization base will have the time defined by the user.

An important part of the visualization component is how it handles the representation of states and links. Every state object has two timestamps, one for the start and other for the end, a value that indicates which of the possible states it represents and a referring general entity. Links have the same information as states, but have additional information to indicate the source entity and the destination entity. A special case of links might be considered when there are several destinations (to represent a broadcast, for instance), but this can be also defined as a set of links objects with the same origin but different destinations.

Figure 3.10 shows a schematic representation of how the visualization component handles states and links to create them in the representation. In this Figure, there are two entities that were placed in the visualization base. Based on the referring entities of state and link visual objects, the visualization component defines their position in the visualization base. In the example, we have two states and one link. The link represents a communication between them. After defining the position in the visualization base, the component obtains the timestamps of

the visual objects to define their size in the timeline.

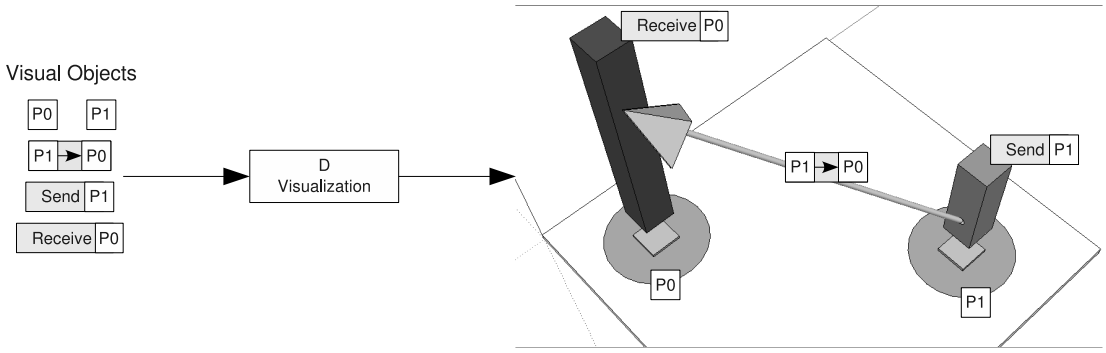


Figure 3.10 – Representation of State and Link Visual Objects in the 3D scene.

Another characteristic of the 3D scene is the visualization base. As previously discussed, we created three different configurations that are rendered in the base. Next Section details how the three different cases generated by the entity matcher are rendered in the 3D scene. The Section 3.6.2 presents the possible interaction mechanisms that can be applied in the 3D scene.

3.6.1 Rendering the Visualization Base

Figure 3.11 shows how the communication pattern is rendered in the visualization base. As input, the visualization component (D) has on its left the visual objects, which are composed of links and entities in this example, and on its bottom the communication pattern generated by the entity matcher. On the right of the Figure, the scheme shows how the visualization of the communication pattern on the base is rendered. The vertical bars are the states of the processes through time.

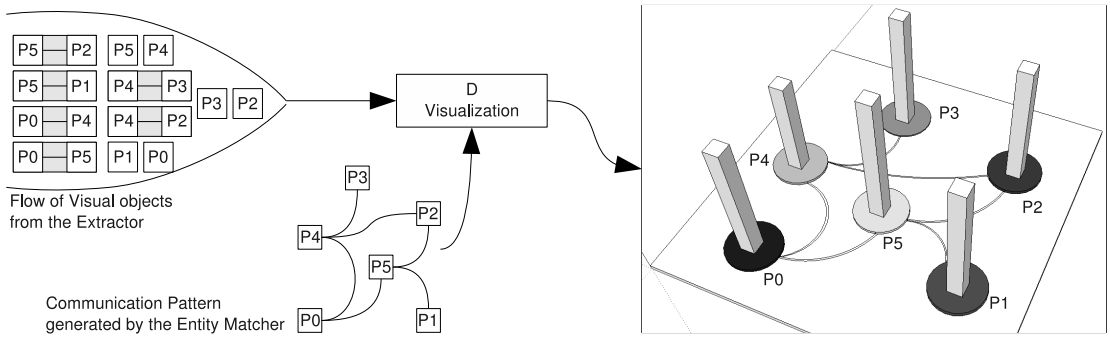


Figure 3.11 – The representation of the communication pattern in the 3D Scene.

Still on Figure 3.11, we can notice that the links among the processes are undirected. In real situations, the trace data can have information about the origin and destination of a certain communication. This data, together with the set of other communications may enable a more complete representation of the communication pattern. The visualization component is able to

enhance the definition of the positions for every process trying to avoid crossing links, improving the perception and understanding of the communication pattern. Another possibility appears when there are several communications between two processes for a given interval of time. The visualization component, in these cases, can generate a visualization where the width of a connection in the visualization base will be larger for pairs that communicate more.

Figure 3.12 shows the second configuration of the entity matcher, composed of the network topology and the communication pattern. The component has as input the flow of visual objects, on the left, and the network topology (represented by the darker and larger lines) on the bottom. The 3D scene is on the right, with the visualization base holding the network topology and the communication pattern. The states represented in the timeline are in the Figure only for information purposes. The links were not drawn in the schematic 3D scene.

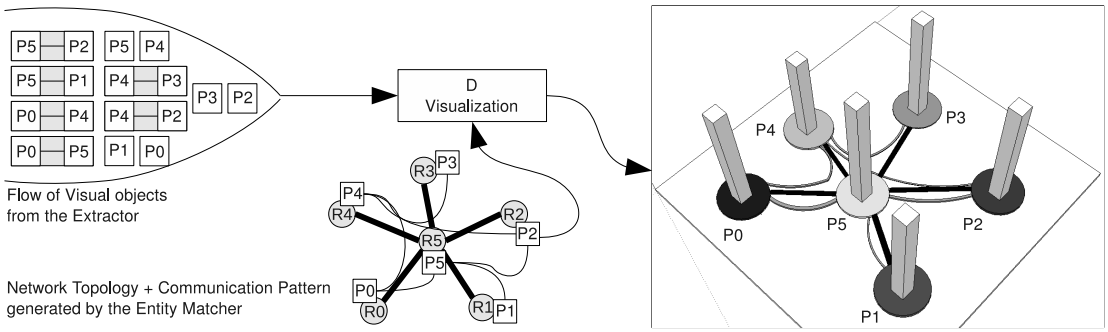


Figure 3.12 – The representation of the network topology and the communication pattern in the 3D Scene.

The second configuration for the visualization base (Figure 3.12) is especially important when network-bounded parallel applications are analyzed. In these cases, the representation can be improved with additional information such as the bandwidth and latency for each link. This combination of characteristics from the network may help the detection of possible communication bottlenecks caused by extreme utilization of one network link, for instance. The representation in the base can be altered to show larger width for network links with higher bandwidth, and different colors to represent latency information in a given time. If routing information is also present, the user may observe which path the messages took during the execution, enabling the analyst to view if an alternative deployment of process would result in benefits in terms of execution performance.

The representation of the third configuration of the base is depicted in Figure 3.13. The logical organization of the components, generated as a hierarchy and represented with a treemap by the entity matcher, is drawn on the base by the visualization component. The resulting scene appears on the right of the Figure. As previous configurations, the representation includes the states representation in the timeline just to show a view of what the 3D scene would look like. Links in the visualization base were removed from the example in order to focus on the treemap generated by the entity matcher. This representation serves mostly as statistical summaries of the application that are rendered in the same scene that detailed behavioral events. The rectangles in the base, that normally represent resources, can be calculated following several characteristics of

the application behavior, such as the number of communications, their size, the amount of time in a given state and combinations of these. The work of customizing this representation to different needs must be done through a cooperation between the entity matcher and the visualization component, since the former has hierarchical information about the organization of the resources and the latter has timestamped visual objects, such as states.

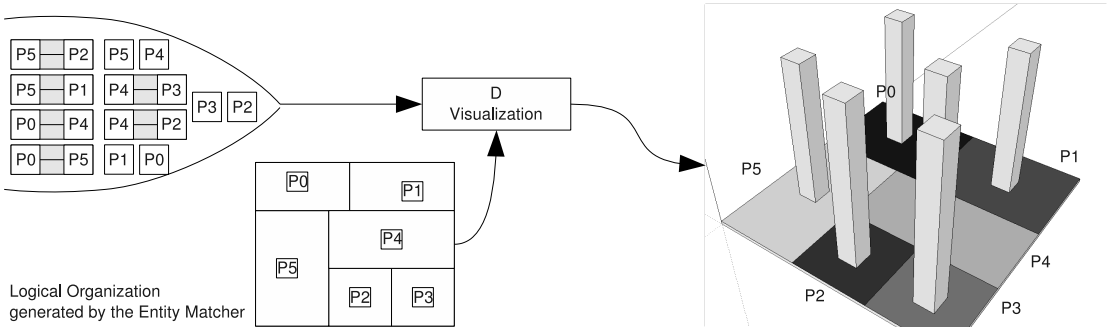


Figure 3.13 – The representation of the hierarchical logical organization in the 3D Scene.

The rendering of the treemap in the visualization base has some peculiarities that must be taken into account. The first one is related to the size of the main square used in the representation. This size is usually defined by the user, but in cases where an increasing number of resources is present, it would be interesting to see the size of the main square increasing automatically. Considering that the 3D space is unlimited, this size could become too big to generate an easy understanding of the representation. To solve this situation, aggregation and reduction mechanisms should be used to downscale the quantity of data that is drawn. The aggregation mechanism that is presented in next Chapter could be applied here.

Another characteristic of the treemap visualization base is when the squares represent machines, for instance. If there are too many processes in the same machine, the visualization will result in a larger number of processes that must "fit" in a given square. If the square is too small, the resulting alternatives are either to aggregate the processes in one entity, or to increase the size of the main square of the treemap. Both alternatives have their drawbacks and benefits and must be balanced to provide an aesthetic visualization to the end user of the 3D representation.

3.6.2 Interaction Mechanisms

The 3D visualization also comes with a number of different interaction mechanisms. Some of these mechanisms were already discussed in Section 3.1. Here, we investigate a step further by giving more details and exploring some examples. First of all, we must first remind of the notion of camera inside the scope of the 3D representation. The visual conception of the 3D approach, described in this Chapter, expects the presence of a camera. This artifact must be present because it is from this viewpoint that the visualizations are created.

Different mechanisms can act on the camera. The first and more relevant is translation operations inside the 3D space. The translation of the camera position allows the camera to go forward and backward through time, for instance. Besides, the camera can also be rotated in the

three axis to give the analyst other viewing angles. Figure 3.14 shows how these mechanisms act to provide different points of view. The first image at left is a replica of the image depicted in Figure 3.12. Subsequent images to the right show the point-of-view from different angles of the same scene.

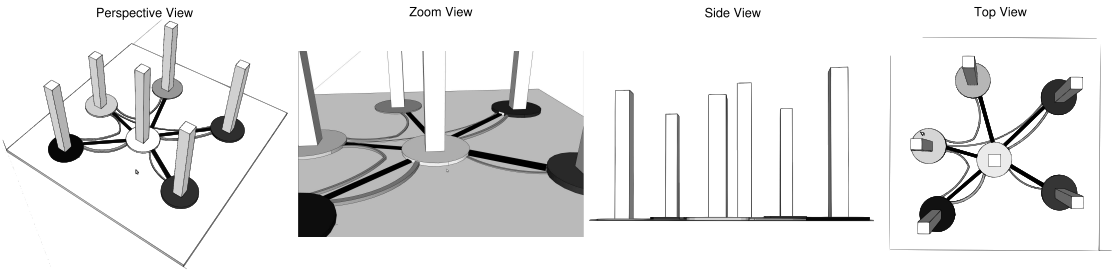


Figure 3.14 – Different points of view of a 3D scene, generated with camera translation.

Other possible interaction mechanisms of the 3D approach is the use of animations and replays. Animations can be used to give the analyst the possibility to analyze the chain of events step by step, viewing the representation of every event one at a time. The dynamic of the animation can also help the observation of repeating patterns during the events evolution. These animations can be combined with the replay technique, showing again specific intervals of time.

Classical interaction mechanisms already present in other visualization tools can also be applied in the 3D approach. Zoom, for example, can be applied by changing the time scale rendering in the timeline, allowing a more detailed analysis when zoomed, and general views of the whole scene when the user has a more significant time slice rendered in the scene. The changes in the time scale can also lead to performance improvements in the way the visual objects are stored. In general views, much of the details that are rendered could be discarded without losing the major understanding of the events.

3.7 Summary

The Chapter has presented the visual conception of the 3D model, explaining the meaning of the three dimensions and the definition of the visualization base and timeline. The proposed model tries to solve the lack of a visualization technique that is able to show application behavior together with network characteristics. We made a step further through a general approach that can show two combinations of representation techniques. The first is the mix between **behavioral** and **structural** representations, that solves the previously cited problem of analysis of application behavior with the network topology. In the context of our 3D approach, the behavioral representation consists in the visual objects rendered along the time axis, and the structural representations are the communication pattern and the network topology rendered in the visualization base. The second combination is between a **behavioral** and a **statistical** representation, the later being the treemap shown on the visualization base. We also have presented the abstract component model that is able to generate a 3D representation. The subsequent Sections are dedicated to the description of each component of the model: the trace reader, the extractor, the

entity matcher with its three sub-components, and the visualization component. We believe that the proposal of the 3D approach can be a viable solution to enable the performance visualization of parallel applications that takes into account the network influence during the execution. The Triva prototype, that implements the 3D model, is presented in Chapter 5. Results obtained with the prototype are described in Chapter 6.

The next Chapter describes the visual aggregation model that is developed in this thesis to obtain visualization scalability in the parallel application analysis. One of the main ideas behind this approach is the use of the treemap technique for the representation of aggregated monitoring data. This is in part inspired by the development of the third configuration of the base, which also uses treemaps.

Chapter 4

Visual Aggregation Model

The previous Chapter has presented our proposal to handle the performance visualization of parallel applications that take into account the network topology. As explained, our solution deals with a three dimensional visualization that is able to show the network topology and the behavior evolution of application components.

Another issue related to grid applications is that they can be composed of a large number of processes. Some analysis is already possible with applications composed by thousands of processes [50], but in clusters. Several issues arise in grid environments when analyzing large-scale applications. A first one is the huge quantity of monitoring data that can be generated by grid applications, depending on two factors: the number of monitored entities and the detail of behavior collected for each of those entities. Another issue in the analysis of large-scale parallel applications is the visual scalability [26], which is about the quantity of data that can be displayed in the screen without losing the ability to understand what is represented.

The fact is that the representations provided by visualization tools must also scale in order to analyze big parallel applications. If we consider only the number of monitored entities, we must be able to represent at least a few thousands of processes in the same visualization. A certain amount of details about each of these entities over time have to be present in the visualization in order to analyze the processes. An example of the lack of scalability in the visualization is the space-time representation, where the amount of data that can be represented is limited by the vertical space available in computer screens.

Among the visualization tools reviewed in Chapter 2, Vampir (Section 2.2.5) offers in its space-time view a hierarchical visualization that increases the amount of processes that can be visualized at once. The technique works by aggregating processes's behavior according to a hierarchical representation. The problem of the approach is that the information shown in each level is represented differently, turning out to be difficult the analysis of the Vampir's aggregated views. Other tools, such as Pajé and Jumpshot, for instance, use scrolling mechanisms to deal with the big number of monitored entities. This has a potential negative impact in the analysis since not all entities's behavior are shown at the same time.

Our approach uses time intervals to dynamically create an annotated hierarchical structure that represents the application behavior for that period of time. We also present an aggregation mechanism that can be applied when there are too many monitoring entities to be analyzed in

the same screen. We employ the treemap technique [42] to create a visual representation of the hierarchies. The combination of the Time-Slice technique, the aggregation model and treemaps increases the number of monitored entities that can be visualized at the same time, and allows a direct comparison among their behavior.

The treemap visualization is already used to observe monitoring data from distributed environments. CoVisualize [68], for instance, is a grid visualization tool developed for PlanetLab. The tool uses values such as CPU, Memory and Bandwidth of nodes to render the treemaps. Besides, it can be configured to show also efficiency images, based on CPU and memory, and usage images, based on slices, slivers and nodes according to the terminology of PlanetLab platform. Another example is the visualization of workloads [39], where the values of the represented hierarchies are calculated based on the workloads applied to resources. In both approaches, the time variable is not used and only the visualization of resources state is represented.

This Chapter is organized as follows. We begin with a description that shows that monitoring data can be hierarchically organized. We present then the Time-Slice algorithm responsible for creating an annotated hierarchical structure that represents the program behavior for a given interval of time. The aggregation model is presented, working by merging data by similarity and moving it to upper levels of hierarchical structures. We then present the basic concepts of the treemap visualization, a technique proposed in 1991 to solve the problem of visualization scalability for hierarchical structures, and its application to visualize the output of the Time-Slice technique and the aggregation model.

4.1 Hierarchical Organization of Monitoring Data

Traditional monitoring systems for distributed environments periodically gather data about the behavior of a pre-defined set of entities. This set can contain resources of the computing system, such as processors and memory, and components from parallel applications, like processes and threads. For each entity, several other types of information are also registered, like events for functions calls, or changes in the value of a variable associated with the entity. An example is Ganglia [55], able to collect monitoring data from several computers and for each of them, the level of CPU utilization, input/output, and memory. For Ganglia, the entity is the computer. Other cases, more focused on the application level, are tracing libraries such as JRastro [20], or the VampirTrace tool. In this later case, it results in application traces that register the behavior of processes and threads, which can be identified as the monitored entities. The states for the processes and threads, their events, are the information associated with them.

An important characteristic of monitored entities is that they can be organized as a hierarchy. This organization lists the observed entities as bottom-level nodes, or leaves, leaving intermediary nodes of the hierarchy to group them based on logical or location characteristics. In the example shown in Figure 4.1, the monitoring system collected data from processes and threads. A possible hierarchical organization of these entities is to group the threads by processes and the processes by machines. If the application were executed in a grid environment composed by clusters, the machines could be also grouped by cluster. Additional information about the processes and threads can also be present in the hierarchy, such as the states $Blocked$ and $Running$ below the $Process$ entity, $Created$ and $Join$ below $Thread$.

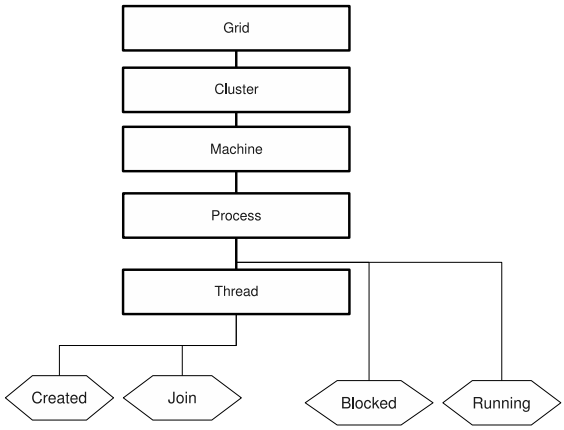


Figure 4.1 – Hierarchy of Entity Types.

Usually, the nodes of Figure 4.1 are types of the monitored entities. The hierarchical structure serves as a guideline to organize the monitoring data collected by a tool that provides such information. During the collection of events about processes and threads, the monitoring system creates instances of these types. Figure 4.2 shows an instantiation of the hierarchical organization, where the application is composed by N_p processes (each with one thread), grouped by N_m machines, N_c clusters, finally all belonging to the same grid.

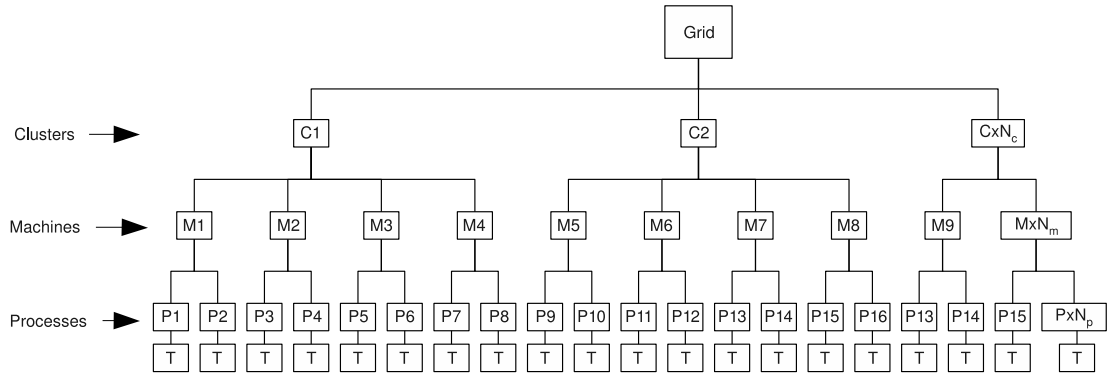


Figure 4.2 – Hierarchy of instances of the entity types.

The types of a hierarchical structure can be related to any kind of entity that can be monitored. If, for example, we are monitoring an object-oriented application, the resulting collected data would be composed by traces from the objects that were instantiated and the methods executed. Another level of the hierarchy is composed of packages that hold the classes. The resulting hierarchical organization would be a tree having as root a type *Package*, with a single child of type *Class* with a child of type *Method*.

The notion of type hierarchy was implemented and validated in the visualization tool Pajé [22]. Its format is considered generic since it can be adapted to represent virtually any kind of monitoring data. It was applied to the visualization of Java Applications [20], MPI applications and

multi-level analysis of parallel applications executed in clusters [70]. One of the reasons for the generic capability of Pajé is the use of a hierarchical definition of the data, being able to adapt to a broad range of monitoring systems, from the ones focused in the analysis of resources to systems used to trace parallel applications.

The type hierarchy of Pajé is enhanced with four additional basic types to describe an entity behavior. They are states, events, variables and links. A state of an entity means that the entity spent an interval of time in that state. An event has just one timestamp and can be used to describe singular events in time. A variable is used to visually describe the evolution of a certain metric over time and a link is used to describe an interaction between two entities. Because these types can describe a broad range of application behavior, we decided to adapt them in the development of the Time-Slice algorithm. This adaptation is described in the next Section.

4.2 The Time-Slice Algorithm

The objective of the Time-Slice algorithm consists in creating a hierarchical structure that reflects the program behavior for a given interval of time. For that, the nodes of the hierarchy must receive values that are calculated based on two factors: the definition of a time interval and a summary of the events for each monitored entity on that time interval.

Different configurations to define the time interval are possible. For example, its length can be changed dynamically in order to find visual patterns from the data being analyzed. This allows the detection of patterns that might appear in a small slices of time but not in larger ones. The user can also move the slice of time being analyzed, allowing the observation of the evolution of the entities through time at a small time scale.

The summary of events is done by taking into account the interval of time specified and additional information about an entity, which is present in the monitoring data. The objective is to find a numerical value that represents the behavior of each entity. There are different ways to define the numerical value for each entity. We can consider, for instance, that this number is the amount of time, or the number of times an event happens, or any other information that can be counted somehow. Before getting into the details of how each of these methods is used to calculate the numerical value, let us proceed to an overview of the variables terminology used in next sections.

Figure 4.3 shows an example where there are two processes, A and B, that have been executed in the machine M, which was part of cluster C and the grid G (hierarchy shown on left of the Figure). The time slice defined for the algorithm begins at T_i and goes to T_f (represented by the two vertical lines). Singular events are denoted by X_{E1}, where X is the identifier for the entity and E the type of the event. The number next to E is a counter to identify uniquely that event. States are defined by $X_{S1}t_i$ and $X_{S1}t_f$, where X denotes the entity, S the type of the state and a number to uniquely identify that state instance. Links have their beginning denoted by $XY_{L1}t_i$ and end by $XY_{L1}t_f$, where X is the origin of the link and Y is the destination. Variables are represented by a series of timestamped events that hold the current value for that variable. The resulting visual representation is denoted by the variable V in the Figure.

In the example of the Figure 4.3, there is one state for the entity A ($A_{S1}t_i$ to $A_{S1}t_f$) and two for the entity B ($B_{S1}t_i$ to $B_{S1}t_f$ and $B_{S2}t_i$ to $B_{S2}t_f$). There are two singular events in the

entity A, denoted by A_{E1} and A_{E2}, and one link ($BA_{L1}t_i$ to $BA_{L1}t_f$). There is one variable for the entity M, denoted by the letter V. We must also define a variable X_{val} that will hold the calculated numerical value for a given X entity.

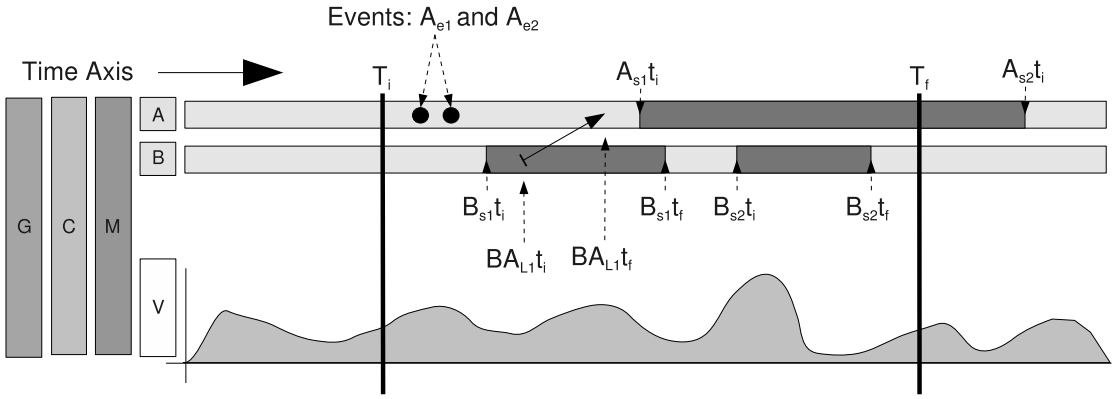


Figure 4.3 – Example showing the mathematical variables used in the algorithm.

The next subsections detail how the algorithm works in the presence of states, variables, links and events. The general principle is to separately sum the values for the each type of state, variable, link and event, and then intersect the obtained value with the time slice used. This Section ends with a complete example of the algorithm.

4.2.1 States

A state is defined by a value and two timestamps, one for its beginning and another for its end. An entity can have states with different values through time. Figure 4.4 shows five entities, from A to E, grouped by location in machines $M1$ to $M3$, and by clusters $C1$ and $C2$. In this example, we use only one value for the state, represented by the darker tone rectangles in the horizontal axis.

For the example of this Figure, the X_{val} values for the entities will hold the amount of time each one stayed in the state in question. There are five different ways to calculate X_{val} for the entities from A to E. These cases are divided taking into account how the state is positioned in time in relation to the selected time slice (T_i up to T_f). The first case is represented in the behavior of entity A (see Figure 4.4), where the value for the entity A_{val} is defined by $T_f - A_{S1}t_i$, because the end of the state is after the end of the time slice. The second case of entity for entity B, the value will be defined by $B_{S2}t_f - B_{S2}t_i$, without considering the amount of time entity B spent in state B_{S1}, since this state is out of the selected time slice. The third case is the entity C, where the state starts before the beginning of the timeline, resulting in the formula $C_{S1}t_f - T_i$. Entity D has no state inside the selected time slice, so its value is simply zero. Entity E has two states within the time slice, we must then consider both to find E_{val}, with the formula $(E_{S1}t_f - E_{S1}t_i) + (E_{S2}t_f - E_{S2}t_i)$.

Considering all these situations and normalizing to the time slice, we obtain:

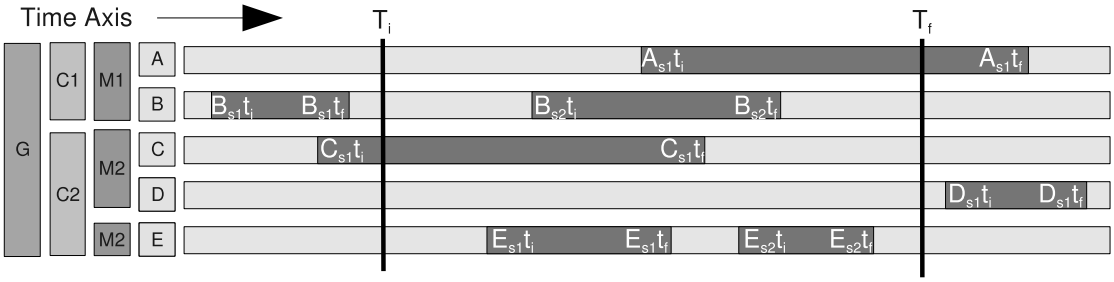


Figure 4.4 – Time-Slice algorithm working to summarize states using amount of time.

$$X_{val} = \frac{\sum_{z=0}^{n}(\min(T_f, X_{Sz}t_f) - \max(T_i, X_{Sz}t_i))}{T_f - T_i} \tag{4.1}$$

During the execution of an entity (e.g. process, thread), it is common to find more than one type of state. When this happens, their values must be calculated separately with the formula. Taking as example the hierarchy of Figure 4.1 with the $Process$ entity, there are two types of states: $Blocked$ and $Running$. The calculation for their values results in $Process_{val-blocked}$ and $Process_{val-running}$. These values are stored in the entity $Process$ like a vector.

4.2.2 Variables

Observation tools gather information about different metrics during the monitoring of a system. Examples of these metrics are the bytes per second transferred by the network card, CPU or memory utilization. They are often collected as events, with different gathering mechanisms. In an ideal situation, monitoring tools must sample metrics using very small time intervals, improving the accuracy of the values collected. The metric $Memory$ in the top part of Figure 4.5 shows how the drawing of the collected values for this metric are in this ideal situation.

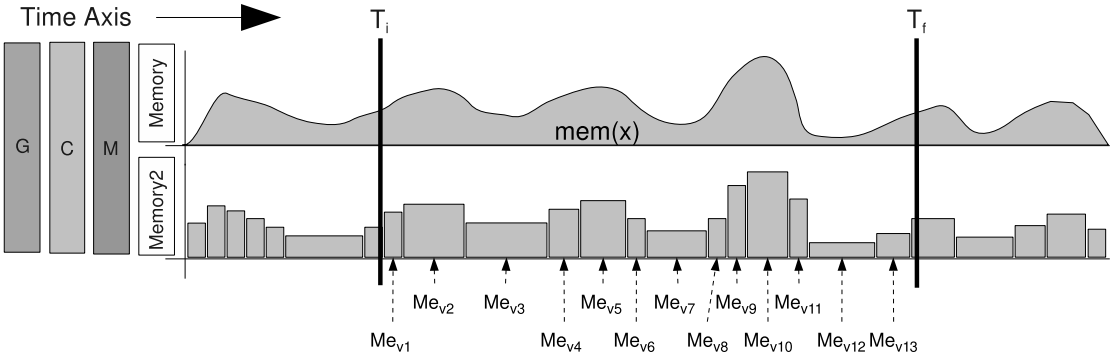


Figure 4.5 – Approximation measurement caused by the frequency of collection mechanisms; the Time-Slice algorithm works using the discrete values collected.

For the ideal situation depicted on top of Figure 4.5, the Time-Slice algorithm performs the integration of the function that defines the value for the metric for that period of time. Considering the $Memory$ variable and the time slice of the Figure, the equation is:

$$X_{val} = \frac{\int_{T_i}^{T_f} mem(x)dx}{T_f - T_i} \tag{4.2}$$

where $mem(x)$ is the function that defines the value of the variable and T_i to T_f is the time slice. In the example of the Figure, the final value for M_{val} is the area of $mem(x)$ limited by the interval of time.

The accuracy brought by the ideal situation in the collection of a performance metric is hard to obtain in the real world. If the gathering system collects the metric value too often, the intrusion caused may lead to a different behavior of the observed system. This behavior might be significantly different from the normal behavior of the system. This can result in the lack of meaning of the monitoring data, since the normal behavior (without observation) is too much affected. To alleviate this problem, and at the same time obtaining a good accuracy of metric's value, monitoring tools use periodic samples between fixed or variables intervals of time. Another perspective for this situation that may solve the problem is an agreement between the collection mechanism and who demands the monitoring data. The agreement can specify the amount of intrusion allowed, or the amount of intrusion obtained when a set of metrics are configured to be collected.

The bottom part of Figure 4.5 shows the metric $Memory2$ and its measured values, inside the time slice, denoted from Me_{V1} to Me_{V13}. Each variable is valid between a defined interval of time: $Me_{V1}t_i$ to $Me_{V1}t_f$, for instance. Considering $Memory2$, the Time-Slice algorithm operates by adding the area of the rectangles. Therefore, the equation used by the algorithm for a more real situation of measurement of metrics is:

$$X_{val} = \frac{\sum_{z=0}^{n} (Metric_z t_f - Metric_z t_i) \times MetricValue_z}{T_f - T_i} \tag{4.3}$$

where $MetricValue_z$ is the value of the metric between $Metric_z t_i$ and $Metric_z t_f$, with n samples collected inside the time slice (T_i to T_f).

4.2.3 Links

Links are used to represent interactions among different entities. Figure 4.6 shows an example where five processes, from A to E, have some interactions among them. A link is denoted by XY_{Ln}, where X is the origin and Y is the destination. If there is more than one link from X to Y, the subscripted number is used to differentiate them. A link can also have a value associated, which is represented by the variable itself. The value can be, for example, the quantity of data transferred. Besides this, a link also has a start time, represented by t_i appended to the variable, and an end time, represented by t_f. As before, T_i and T_f are used to define the time slice.

The way the Time-Slice algorithm works to summarize links is different from states and variables. Instead of simply associating a unique value to the entity, the links are used to create two values. One of them is created when the entity is the origin of the links, and the other

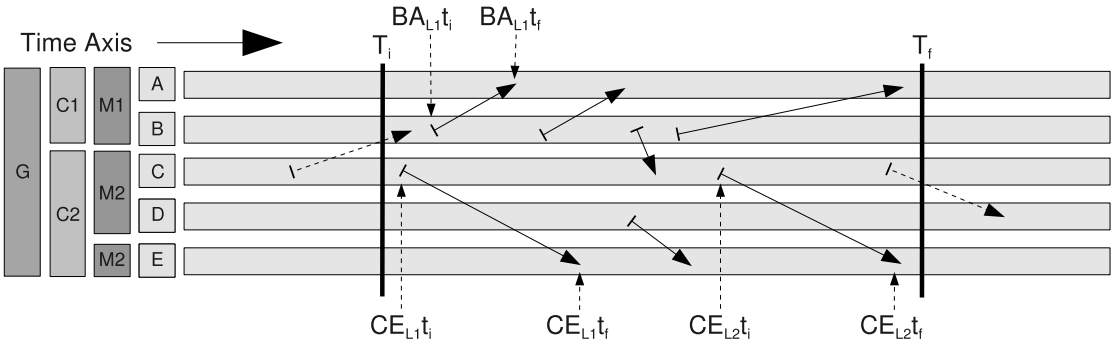


Figure 4.6 – Time-Slice algorithm treating links presence in the time slice using amount of time versus data transferred.

appears when the entity is the destination. Therefore, for an entity X, we define in the following equations $X_{val-as-origin}$ and $X_{val-as-destination}$:

$$X_{val-as-origin} = \frac{\sum_{z=0}^{n} (XY_{Lz}t_f - XY_{Lz}t_i) \times XY_{Lz}}{T_f - T_i} \; for\ any\ entity\ X \tag{4.4}$$

$$X_{val-as-destination} = \frac{\sum_{z=0}^{n} (YX_{Lz}t_f - YX_{Lz}t_i) \times YX_{Lz}}{T_f - T_i} \; for\ any\ entity\ Y \tag{4.5}$$

where XY_{Lz} is the value of the link z of a given entity X, and YX_{Lz} is the value of the link z of a given entity Y. It is important to notice that links that cross the time slice boundaries are not considered here.

Adaptations to these equations are possible in different situations. If we want to view only the amount of time spent by a link between two entities, we can ignore the value attribute of the link in the equation. Another perspective is when we want to view the performance of each link, by dividing the quantity of data transferred by the time it consumed to do the transfer. With this calculation, the value for a given entity matches the performance of the entity's communication either as origin or destination of the links. A third situation happens when we need to know only the amount of data transferred by a single entity. In this case, we ignore the variables of time in the equation. Several other combinations are possible depending on the additional data available in each link, such as overhead for creating the packets and emitting or receiving them and so on.

A special case for summarization of links is to count the destinations, for example, for a given origin. For the entity B of Figure 4.6, for instance, it results in three links with destination A and one link with destination C. This adaptation of the algorithm enables the observation of groups that communicate more intensively in a parallel application.

4.2.4 Events

Events are singular points in the time axis that indicate when something happens for a given entity. They can represent the act of changing the value of a variable, or the reception of a

message. To summarize their existence in the behavior of a given entity, the easiest way is to count them by their type. The resulting value for the entities can be composed of these counts: number of times a variable changed, how many message receptions occurred, and so on. Different adaptations are also possible if additional data is available in each singular event.

4.2.5 More statistics

In previous subsections, states, variables, links and events were detailed separately. In the context of states, we presented the algorithm working with only one state at a time. Additional meaningful statistics can also be extracted when we consider more than one state for a given entity. This situation depends on what the meaning of the states is and how they can be combined. An example for that is the combination of states that mean actual processing and states that mean communication. Their combination can give the analyst a view of the ratio computation/communication for all the entities of the parallel application.

The same techniques also apply to other types of monitoring data, variables, links and events. These combinations depend on what is the nature of the summarized value. Up to now, we have seen that these values can be related to the amount of time (in the states case), accumulated value of a metric (variables case), quantity of data in bytes (links case), simple counts (events case). Additional information that might be present in the monitoring data can also increase the range of possible summarization values. Table 4.1 gives an overview of possible combinations that can be used to obtain more statistics from the basic types of monitoring data.

Table 4.1 – Non-exhaustive set of combinations to obtain more statistics from traces.

Combination	Unity	Application
Bytes per second	Quantity/Time	Communications Performance
Computation vs. Communication	Time/Time	Efficiency of processes
Blocked State vs. Number of Links	Time/Count	Mean time blocked per link
Computing State vs. CPU Utilization	Time/Value	Efficiency

4.2.6 Example

Figure 4.7 shows an example with five monitored entities, from A to E, grouped by their execution machines, represented by the rectangles $M1$, $M2$ and $M3$. The machines are grouped by their clusters $C1$ and $C2$, which are part of the grid G. The selected interval of time is 9 seconds, limited by the two vertical bars (small vertical bars mean intervals of one second). In this example, we intend to summarize three different information: the amount of time of the states $Blocked$ (darker rectangles), $Executing$ (light gray rectangles), and the bytes per second of the links $Communication$ (represented by the non-dashed arrows in the middle of the time slice). The numbers in the beginning of the communications represents the quantity of data transferred, in bytes. The link summary is attributed in this example to the origin entity.

Considering the case shown in Figure 4.7 with two states represented, Table 4.2 lists the values of the entities for the three summaries. The first column shows the five entities; the

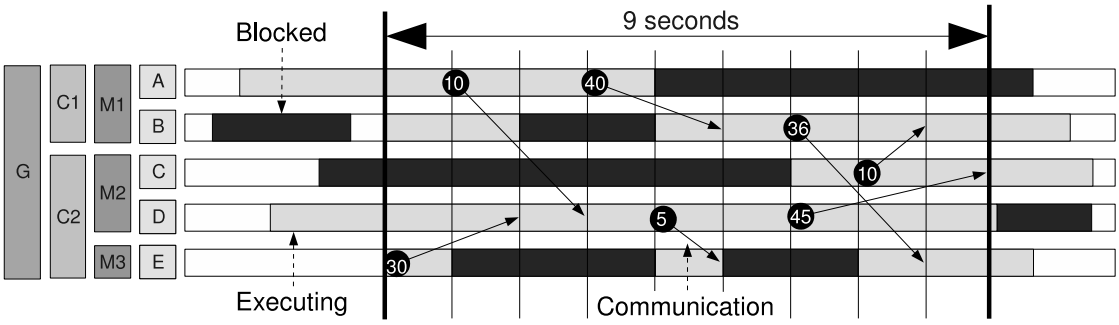


Figure 4.7 – Complete example showing different aspects of the Time-Slice algorithm.

second column shows the time in seconds each entity stayed in the $Blocked$ state within the time slice; the third column shows the time in seconds for each entity in the $Executing$ state; and the fourth column shows the bytes per second associated with each origin entity of the link $Communication$. For instance, to summarize the amount of time of the $Blocked$ state of the entity A, we sum up its duration of 5 seconds that is within the time slice. To summarize the links, we use the bytes transferred divided by the time the origin process took to the transfer. For entity A, the $Communication$ summary must be the sum of $10bytes/2seconds$ and $40bytes/2seconds$, resulting in $25bytes/second$.

Table 4.2 – Summaries for the three different aspects analyzed in Figure 4.7, considering the time slice of 9 seconds.

Entity	$Blocked$ (Time in sec.)	$Executing$ (Time in sec.)	$Link$ (Bytes per second)
A	5	4	$10/2 + 40/2 = 25$
B	2	7	$36/2 = 18$
C	6	3	$10/1 = 10$
D	0	9	$5/1 + 45/3 = 20$
E	5	4	$30/2 = 15$

Figure 4.8 shows three hierarchical organizations of the example of Figure 4.7, considering the three summaries presented in Table 4.2. These hierarchies are the result of the Time-Slice algorithm, representing the behavior of different aspects of the parallel application inside the selected interval of time. The values of the leaves of the structure are defined based on the calculated summaries in a per process fashion.

When different types of events are present in the interval of time selected by the user (as the example of Figure 4.7, with two different states and links), the Time-Slice algorithm creates as output a single hierarchy where the leaves have the calculated values for those types. Figure 4.9 shows the output for the current example, where each leaf node has three values that show the blocked state, executing state and communication link, respectively. These values are the same found on the leaves of the three hierarchies of Figure 4.8.

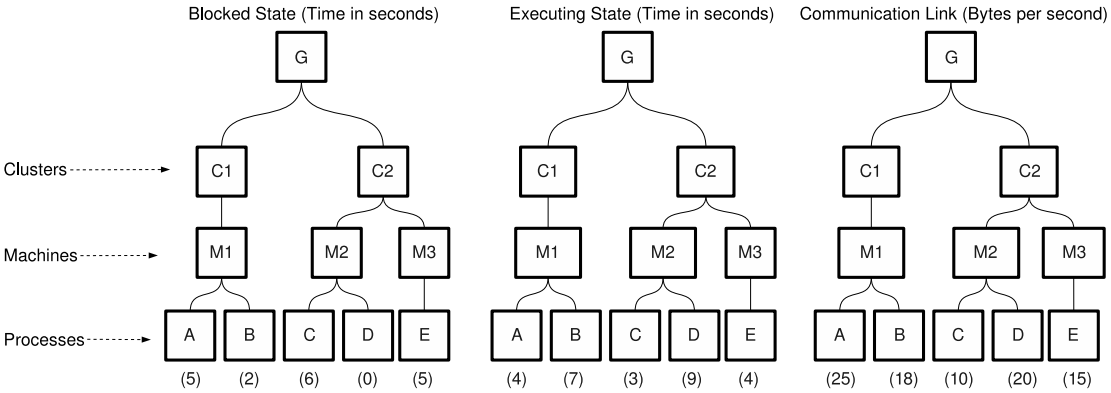


Figure 4.8 – Hierarchical summaries generated by the Time-Slice algorithm considering the three aspects presented in Table 4.2.

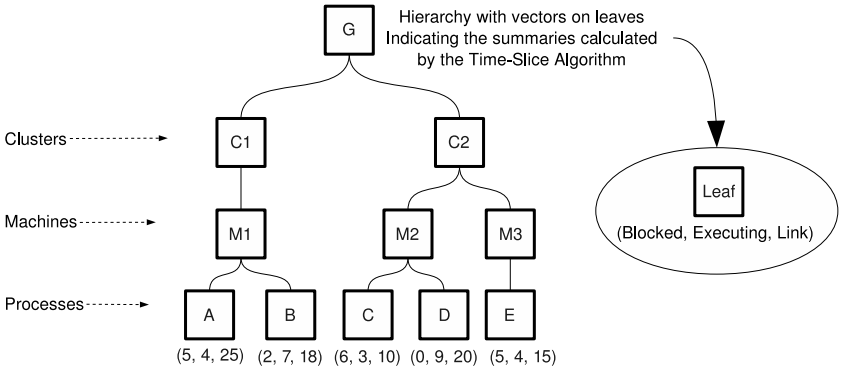


Figure 4.9 – Single hierarchy, based on the ones of Figure 4.8, with multiple summaries on the leaves, generated by the Time-Slice Algorithm.

4.3 The Aggregation Model

Depending on the number of monitored entities present in the traces, the hierarchy generated as output by the Time-Slice algorithm can become too large. If we take as example an application composed by one thousand processes, each one with four threads, the resulting hierarchy in this case will have four thousand leaves. The aggregation model presented here intends to explore the hierarchical organization of the monitoring data in order to provide aggregated values for intermediary levels of the hierarchy.

Figure 4.9 shows the output of the Time-Slice algorithm, represented by a hierarchy with a vector of summary values on the leaves. Considering only the first two values of the leaves's vectors, we obtain the leftmost hierarchy of Figure 4.10. This left hierarchy shows on the leaves the summary value for the $Blocked$ and the $Executing$ states. The Figure also shows three modifications in the hierarchy, caused by the aggregation model. In the example, there are three intermediate levels: Process (P), Machine (M) and Cluster (C). The main goal of the aggrega-

tion model is to group the summary values of a level to the level immediately higher. Therefore, after the first aggregation, the values of the processes in the same machine are added and attributed to the machine node. The algorithm can be applied again to pursue the aggregation, up to the root level, as shown with the second and third aggregation steps of the Figure.

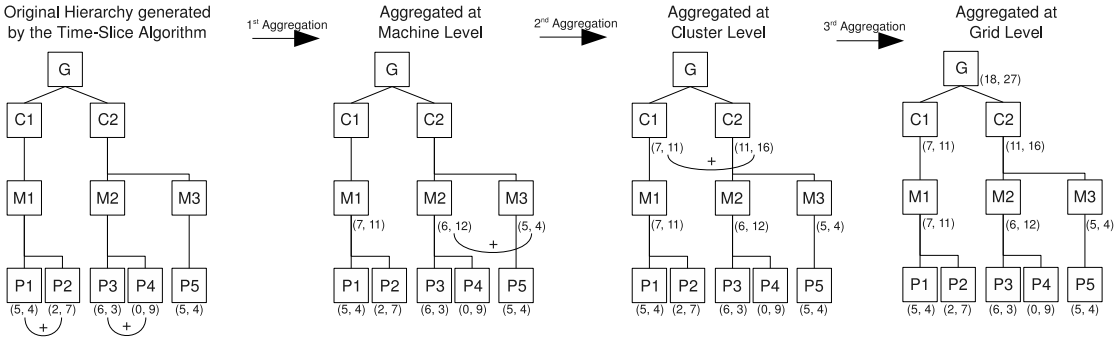


Figure 4.10 – Three aggregations to decrease the hierarchy depth and improve the final visualization with treemaps.

After applying the aggregation model, the intermediary nodes up to the root node have values that were calculated based on the leaves of the tree. The resulting aggregated tree, shown on the right of Figure 4.10, enables a per-level analysis of the data. Since the summary values of the nodes of this tree are the $Blocked$ and $Executing$ states, an analysis in the cluster level, by observing nodes $C1$ and $C2$, enables the conclusion that for the considered interval of time, the cluster $C1$ stayed 7 seconds in the $Blocked$ state and 11 seconds in the $Executing$ state. The same conclusion can be made for the cluster $C2$ and to other intermediary nodes, such as the ones of the Machine level. When there are too many leaf nodes, the analyst can choose to observe only up to a level, avoiding too many details and still being able to understand the overall behavior of the parallel application for the considered time slice.

4.3.1 Aggregation Functions

Besides the traditional addition operation (shown in Figure 4.10), the aggregation model can be applied with other functions to aggregate values, such as max, min, and average. Their direct application depends on what type of value is attributed to the leaf nodes of the original hierarchy and can highlight particular characteristics when aggregating data.

The search for low-throughput communication links, bad transfer rates among processes, small amount of time spent with calculus, for example, can be eased by using a min function when aggregating data. The application of this function can highlight, during the aggregation, the part of the machine that contains the worst communication links, or transfer rates, for instance. On the other hand, a max function can be applied in the aggregation if the user searches for highest values, such as bigger amounts of time spent to calculations, or transferred data.

4.4 Visualization of the Approach

The previous Sections have detailed the Time-Slice technique and the aggregation model. Taking into account an interval of time, the Time-Slice technique works by summarizing different aspects of the monitoring data and creating a hierarchical structure that represents the behavior of the parallel application for that time slice. The aggregation model works by calculating values for intermediary nodes of the hierarchies generated by the time slice. There are several ways of creating a visual representation of a hierarchical structure. This is what the node-link representation does to create Figures 4.8 and 4.9.

Instead of using these classical node-link representations for the output of the Time-Slice algorithm, the work presented here explores the Treemap technique [74] in order to visually represent the created hierarchical structures. The main benefits of this technique are its scalability to show large and deep hierarchies, and the fact that all the screen space is dedicated to its representation.

The next subsection details the basic concepts of these hierarchical representations, exploring more extensively why we have decided to use the Treemap technique. After this, we discuss the scalability issues related to the treemap representation and how the aggregation model can be used to improve the work on this matter. We end the Section showing how the treemap is used to create a visual representation of the hierarchies created in the example of previous Section.

4.4.1 Treemaps Basic Concepts

The traditional way of displaying hierarchical data is to use node-link diagrams [61]. This representation is depicted in the leftmost part of the Figure 4.11. These diagrams are easy to understand by explicitly showing the relation among the nodes. The problem with this approach appears when we try to visualize large scale trees with thousands of nodes. This happens mostly because they do not exploit well the screen space [74].

The treemap technique was proposed in order to solve the scalability problem of hierarchical representations [74]. Instead of drawing nodes and links between them, it uses the whole screen space with a space-filling algorithm. This algorithm recursively divides the space dedicated to draw the hierarchy, following the tree organization. The right-part of Figure 4.11 shows an example of the steps performed by the treemap algorithm to create a representation of the hierarchy shown on the left. For this example, we consider that each leaf node has a value of one, so their sizes are the same in the final Figure. The parent nodes *A*, *B* and *C* have their values, 6, 3 and 2 respectively, defined based on their children. The algorithm starts by the root node A, represented in the middle of the Figure as a big square. The algorithm recursion goes to the second level, dividing the space of node A among their children B, C, and D. Then, the third level is considered, dividing the space of B among its children: E, F, and G; and the space of C, between H and I. The final representation is depicted on the right square of the Figure. In this simple example, the hierarchy is highlighted with the use of margins between inner and outer rectangles in the representation. The presence of these margins depends on the importance of the hierarchy during the analysis. Sometimes they are not present to avoid the loss of pixels of the screen that can be better used to show real data. Figure 4.11 shows a peculiar example. In the general case, the sizes of leaves are not always the same.

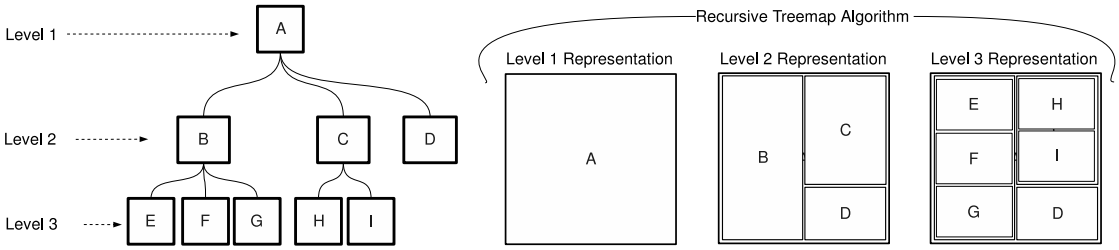


Figure 4.11 – Two types of representation of hierarchical data: the classic node-link diagram and the corresponding treemap technique applied to the same hierarchy.

The treemap algorithm has passed through several evolutions since its creation. One of them is called the Cushion Treemaps [78], a technique that works with the association to each rectangle of an intuitive shading that improves the user perception of what is being shown. Another work based on the original technique is called Squarified Treemaps [13]. It manages to keep the rectangles shapes as close as possible to squares, making the visualization of the information easier by avoiding rectangles with a big width/height ratio. Another proposal called Ordered Treemaps [73] tries to keep nodes proximity when zooming at different levels. Voronoi Treemaps [9] is a different approach to visualize hierarchical data that uses polygons to represent nodes, instead of the traditional rectangles or squares. The polygons are constructed from median lines between pairs of points.

Examples of treemaps utilization include network security [54], grid resource monitoring visualization [68], visual analysis of stock market [77] possibly applied to a million of items [28]. These multiple applications of the treemap technique, including the possibility of showing big hierarchies, motivate us to use it in the analysis of parallel applications. The principal advantage of the treemap representation is the good use of screen space, correlating screen space with the values of the nodes of large-scale hierarchies, and outlining the repartition of this space. On the other side, the drawback is that the hierarchy is less apparent and easy to detect, turning out to be difficult when first analyzed. The benefits of the treemap, however, are more evident than its drawbacks, since the representation can be interactive to allow an easy highlighting of the hierarchy when necessary.

4.4.2 The Scalability Issue

The main advantage of the treemap technique is its ability to draw in an understandable way large-scale hierarchies. This is possible because it involves a space-filling algorithm that uses all the screen space available. If we compare treemap abilities to traditional node-link representations, the scalability of the approach is even more obvious.

Although scalable, the traditional treemap technique is limited by the size of the screen space dedicated to its representation. If the hierarchy being represented is composed by a large number of nodes, the space-filling algorithm may generate squares that are too small. If we consider a computer screen with a resolution of 1024 pixels in the horizontal dimension and 768 pixels in the vertical dimension, we end up with a total of 786432 pixels to be used by the

treemap algorithm. Considering that each square size reasonably occupies at least 100 pixels (10x10 square), the maximum number of leaf nodes of the hierarchy being represented is 7864. Furthermore, if we want to represent at least 2 different states (executing and blocked) at same time, we end up with a drawing that may deal at most with 3932 processes. Today, it is not difficult to find parallel applications larger than that, especially if we want a visualization of threads along with processes behavior.

The visualization scalability can be achieved with the treemap technique by letting the algorithm work only up to a certain level. Therefore, if the hierarchy is composed of many leaf nodes, they are ignored in the representation. This solution is also recursive, starting from the root level, and making it possible to limit the representation depending only on desired depth.

The problem with this approach is that some part of the information that is on the leaves is lost. An example of this is depicted on Figure 4.12, which takes as input the hierarchy generated by the Time-Slice algorithm present in Figure 4.9, only with the summary values for the $Blocked$ and $Executing$ states. Since this hierarchy did not pass through the aggregating model, the intermediary nodes do not have aggregated data about the states. They only have the added value of the nodes below it For instance, $P1$ has a value of 9, which is the sum of 5 and 4; $M1$ has a value of 18, which is a sum of the values of $P1$ and $P2$; and so on up to the root node. This information is necessary to the treemap algorithm, since it expects for each node of the tree an associated value that indicates how much space of the screen that node will take during the representation. The vectors of the leaves represent the amount of time each process, from $P1$ to $P5$, stayed in the $Blocked$ and $Executing$ states. The right part of the Figure shows different treemaps for which rendering was limited to a given level of the hierarchy. The rightmost treemap, on the bottom, actually shows the states for all the processes. It may have on this level squares that are too small in situations with a large number of nodes involved. If this happens, the treemap algorithm may be stopped in a higher level of the tree. The Figure shows, through the others treemaps, that information is lost if this happens. The lost information in the example is the partition of time between each state for each process.

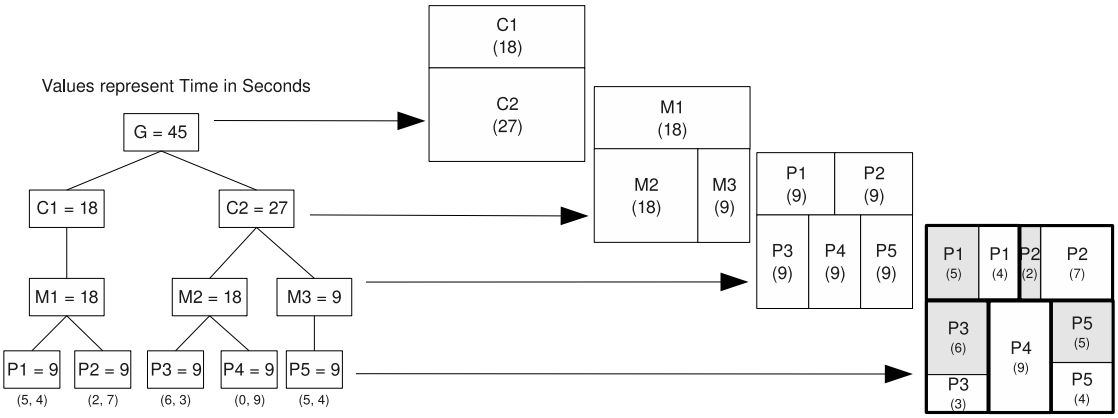


Figure 4.12 – Limiting the treemap representation up to a certain level of the hierarchy to obtain visualization scalability.

The aggregation model proposed in previous Section tries to achieve visualization scalability

through the use of treemaps without losing information that may be on leaves of the represented hierarchies. As presented, the model works by merging data from one level of the hierarchy and moving the resulted merged data towards the root of the tree. The next subsection describes treemap representations generated with hierarchies created with the Time-Slice algorithm and the aggregation model.

4.4.3 Using Treemap in the Example

First, let us proceed to treemap representations of the hierarchies created with the Time-Slice algorithm, without any aggregation. The hierarchical structures of Figure 4.8 are sent to the treemap algorithm. Its drawing procedures take into account the values for each of the nodes in order to generate the maximum utilization of the screen space dedicated to represent the structure. The results of these drawings are depicted on the three different treemaps of Figure 4.13. The left-most treemap was constructed taking into account the hierarchy that defines the behavior for the $Blocked$ state of the processes from A to E. The area of each rectangle represents the amount of time in seconds that each process stayed on that state. Below the main treemap drawing at the left of the Figure, there are three smaller representations that show the summarized view for each level of the hierarchy. We can also use these representations to make higher-level comparisons among the resources that contributed to the application execution.

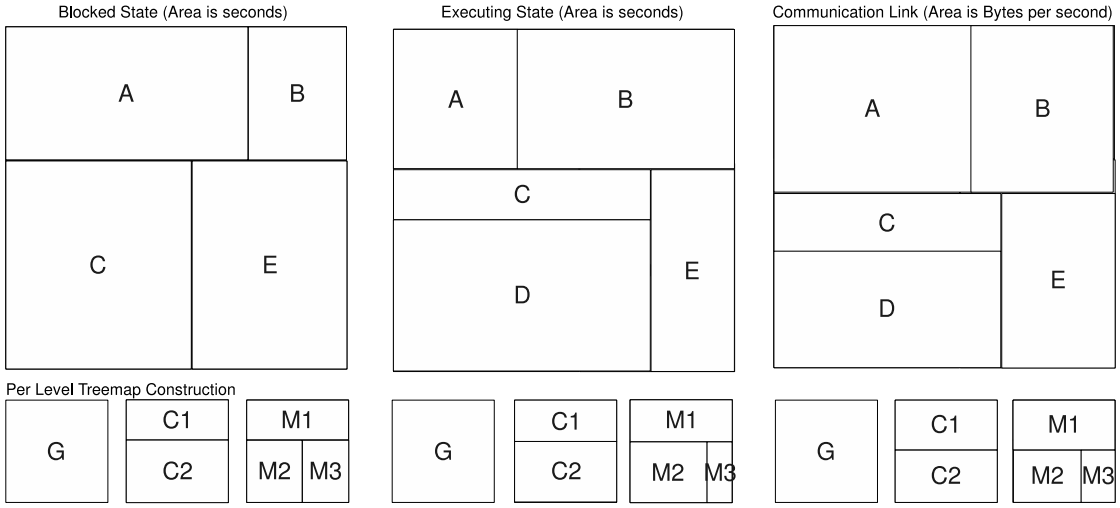


Figure 4.13 – Treemap representations for the hierarchies depicted on Figure 4.8.

The center treemap of Figure 4.13 shows the behavior of the processes for the $Executing$ state. It was built based on the center hierarchy of Figure 4.8. We can see through this representation that process B and D stayed more than others processes on the executing state. Taking into account the smaller treemaps below, we can see also that machine $M2$ contributed more to the execution, when compared to machines $M1$ and $M3$.

The last treemap, on the right-most part of Figure 4.13, shows the representation for the bytes transmitted per second among processes on the selected time slice. The analysis of this

Figure enables the observation of which process obtained a higher throughput.

Generally speaking, the Time-Slice technique presents quantitative data in a more synthetic way. This means that the user can visually and almost instantaneously compare the size of all rectangles. Analyzing the treemaps of Figure 4.13, we can easily see which process has spent more time than others on each particular state. If this representation is used to analyze parallel applications behavior and the state is a blocking operation, the visualization will show which processes spent more time blocked than actually executing. Other types of states and events from the application can be taken into account and combined in the same visualization.

Another characteristic of the representations of Figure 4.13 is to draw the treemap using only available values up to a certain level of the hierarchy. This is depicted in the smaller treemaps at bottom, showing the representation of level $Grid$, $Cluster$ and $Machine$ for each case. These per-level views allow an analysis with less details when a considerable amount of data is present in the deepest level of the hierarchy, maintaining the representation understandable even with a higher number of processes to analyze.

Aggregated hierarchies generated by our aggregation model can also be represented with treemaps. Figure 4.14 shows the treemap visualizations that are generated based on the hierarchies of Figure 4.10. The left most treemap shows the visualization of the original hierarchy, with $Blocked$ (represented by the letter B in gray areas) and $Executing$ (represented by E in white areas) squares being grouped according to the processes. The dashed circle shows the area that corresponds to process $P3$. In this first treemap, the rendering is performed taking into account the values of the Process level of the hierarchical structure. The size of the areas marked by B and E are based on the vector values of the nodes. The aggregation algorithm group these values according to the machines, cluster and the grid. The second treemap of the Figure shows in a comparable way the B and E values for each machine. These values are calculated based on the ones defined for the processes of each machine. The dashed circle in this case highlights the area for machine $M2$. The other two treemaps to the right shows the aggregated view of the values according to the cluster level and the root level.

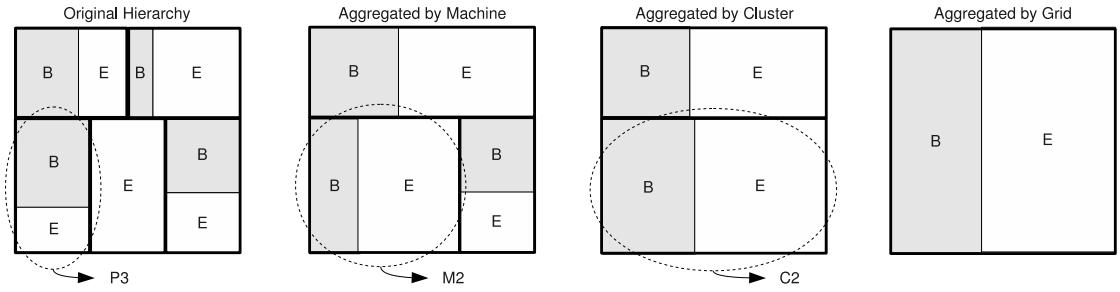


Figure 4.14 – Treemap visualizations based on the original and aggregated hierarchies presented in Figure 4.10.

4.5 Summary

Large-scale parallel applications that run on parallel and distributed architectures exist today, being composed of thousands of processes. These applications need to be analyzed in terms of performance and resources utilization. The lack of visualization tools that can adapt to the large-scale characteristics of these applications motivated the visual aggregation model.

The Chapter has started with a description of the hierarchical organization of monitoring data, a pre-requisite to the model itself. Then, we have presented the Time-Slice technique, which works by summarizing the behavior of a parallel application in a time interval. The output of this technique consists in an annotated hierarchical structure, that serves as input to the aggregation model. Basic concepts of the treemap representation have also been presented, together with its application to visualize the hierarchies generated by the Time-Slice technique and modified by the aggregation model.

The next Chapter details the implementation of this technique, and the three-dimensional model described in the previous Chapter, in the Triva prototype.

Chapter 5

Triva Prototype Implementation

The last two Chapters have presented the visualization models developed in this thesis: the 3D visualization, focused on the highlight of the network topology in contrast with parallel application's processes; and the Time-Slice algorithm with its aggregation model and the treemap visualization. Those Chapters described the models from a theoretical point of view.

This Chapter describes the developed prototype in order to implement the visualization models proposed. The description here details the software decisions taken during the development and the internal algorithms of the implementation. The prototype is named Triva, to stand for ThRee dimensional Interactive and Visual Analysis.

One of the main guidelines to implement the prototype Triva is to build it on top of existing tools and libraries, mainly to avoid the re-implementation of already validated implementations. The first decision following this guideline is the adoption of some parts of the visualization tool Pajé. The main reasons that motivated its adoption are listed in the next Section. This includes a description of the most important components regarding Triva and a performance evaluation of the Pajé Simulator. Other decisions considering software re-use appear in other parts of the Triva prototype. They relate to the input data, the file format used to describe resources, the rendering calculation of graphs of network topology, and so on.

The rest of this Chapter is organized as follows. After the description of Pajé, we present the Triva architecture and how the implementation components are organized. Details about the architecture are presented in three parts: input, the 3D-based and the treemap-based visualization. There is one Section to describe each one of these categories. We end the Chapter with a summary that lists the main decisions about the implementations of the Triva prototype.

5.1 Using the Generic Visualization Tool Pajé

Pajé is a generic visualization tool that has characteristics such as extensibility, interactivity and scalability. The architecture of the tool, depicted in Figure 5.1, is composed of a set of interconnected modules and filters. There are modules that deal directly with the arrival of trace data from trace files, shown on the left of the Figure. These are the FileReader and the EventDecoder. Their responsibility is to convert the events in the Pajé file format to internal objects used by the tool. The trace data, after this transformation, follows the path through

the PajeSimulator up to the StorageController, where it is stored in memory in scalable data structures.

The PajeSimulator is the main part of the tool, since it simulates the behavior of the traced parallel application with real traces. As result, it generates high-level, generic and abstract objects that are called Pajé objects, detailed in the next subsection. The Figure 5.1 also shows the set of possible filters that can alter the flow of Pajé objects towards the two visualization modules on the right: the SpaceTimeViewer and the StatViewer. More details about Pajé's visualization techniques are reviewed in Chapter 2.

One of the main filters of Pajé is the AggregatingFilter. The filter is responsible for reducing the amount of information in a given container based on the level of zoom currently being used by the analyst. The filter, when used, can increase dramatically the scalability and interactivity of the tool by giving fast response to the queries of the visualization components. Another component that is important in Pajé architecture is the PajeTraceController, depicted on the bottom of the Figure 5.1. It controls the initialization of the modules and the appearance of the menu with several options offered to the parallel application analyst.

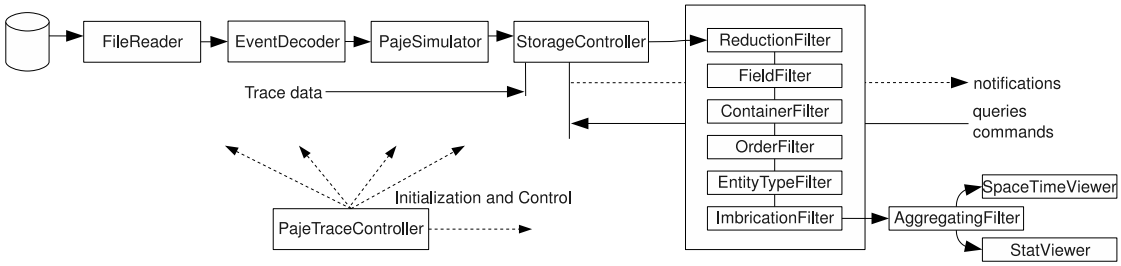


Figure 5.1 – Pajé Architecture.

The components of Pajé use a protocol, composed of notifications, commands and queries. As depicted in Figure 5.1, notifications go from the StorageController through the filters to the visualization modules. These notifications mainly announce changes in trace data, such as modifications in the trace structure or the presence of new information. Commands and queries go from the visualization components to the StorageController. Commands are forwarded to filters to change their behavior and are generally triggered by user interaction, such as the configuration of a given filter by Pajé's graphical interface. Queries, on the other hand, are responses to notifications and are generated by visualization components to obtain information about the traces. A typical query is the request for events information for a given time frame, that is eventually drawn in the visualization window of Pajé. The queries and their respective responses navigate through the set of filters. If a filter is properly configured, it can act on the data changing its information content that will be returned to the query's origin.

The next subsection presents notions related to the type hierarchy and the Pajé objects. Afterwards, we present a performance simulation experiment with Pajé to test the scalability of the tool. An analysis of the obtained results with the main advantages and disadvantages of Pajé adoption in the Triva prototype are presented in subsection 5.1.3.

5.1.1 Type Hierarchy and Pajé Objects

As stated in previous Section, Pajé is a generic visualization tool. This means that it can be used to perform analysis of a wide spectrum of situations. Initially conceived to visualize parallel and distributed applications, the generic capability of Pajé is achieved by using abstract types that can be adapted to any kind of data. There are five types in Pajé: container, state, variable, links and events.

A container type is the only type that contain other types, including another container type. It has an identifier and a name, and usually a start and an end timestamp. All other types must be enclosed within a container. A state type may be used to represent that a given container type can remain in a given state for an amount of time. A variable type usually represents a metric which value changes through time. A link type is used to represent interactions between two container types; and an event type is to mark something that happens in a point in time.

Besides the events produced by the monitored system, a Pajé trace file [23] must also have the definition of the type hierarchy for that file. A type hierarchy is a hierarchy formed by the definition of containers, states and so on. An example for that might be a type hierarchy that reflects the monitoring of parallel applications composed by processes and threads. In this example, the type hierarchy has a container type **process**, that has a state type to indicate the state for that process, and a sub-container type **thread**, also with a state type to indicate the possible states related to that thread. Other information can be defined using the event, variable and link types to reflect the behavior for that application. The terminology of Pajé types is used in next Sections extensively.

Considering the presence of a type hierarchy in a Pajé trace file, the subsequent events must instantiate the defined types, with the creation of containers and the attribution of values to states, links, variables and events that might be present on the type hierarchy previously defined. When treated by the Pajé Simulator component, these events are transformed Pajé Objects, which are generic representations of the events present in the trace file. These objects can be generically treated by the filters and components that are connected at the output of the Pajé Simulator.

The overall organization of a Pajé trace file is composed of three parts: the declaration of events used in the file; the type hierarchy and the timestamped events. In the first part, all the events that can be found in the trace file must be declarated. The lines starting with % of listing 5.1 shows the declaration of the event $PajeCreateContainer$, with its unique identifier – 4; and the rest of its fields: Time, Alias, Type, Container and Name. The other lines show an example of use of this event, appearing usually in the third part of the Pajé trace file, after the declaration of the type hierarchy. The first of these lines indicates that in time 0.1, a container of name “Site Nancy” is created with the alias $Nancy$. The other two lines indicate that in times 0.2 and 0.3, two containers are created: $Grelon$ and $Grillon$, both inside the container $Nancy$. More details about the Pajé trace file, including all other events, can be found in [23].

Listing 5.1 – Declaration of the PajeCreateContainer event.

```
%EventDef PajeCreateContainer 4
%       Time date
%       Alias string
%       Type string
%       Container string
%       Name string
```

```
%EndEventDef

4       0.1     Nancy   0       0       "Site Nancy"
4       0.2     Grelon  1       Nancy   "Cluster Grelon"
4       0.3     Grillon 1       Nancy   "Cluster Grillon"
```

5.1.2 Simulator Performance Evaluation

As stated, the Pajé components transform the trace data into higher-level objects. Among the components, the one that plays a key-role in this transformation is the PajeSimulator and the StorageController. We perform a set of performance tests in order to assess the scalability of these components when the number of entities present in trace files increases. This performance evaluation has been performed both in terms of execution time and memory use.

A measurement tool was implemented to conduct this performance evaluation. Figure 5.2 shows the overall organization of the tool, where the white components are from Pajé and the gray rectangle indicates the implemented module. The FileReader component of Pajé has the definition of the chunk size, which gives the amount of data that will be read at once by the component. For our performance tests, Pajé was configured to have a chunk size of 500 megabytes. This was necessary to avoid multi-chunk file read overhead that might influence a part of the obtained results. Since the largest trace file we generated for the tests is less than 500 megabytes, all measurements are conducted with the same software behavior.

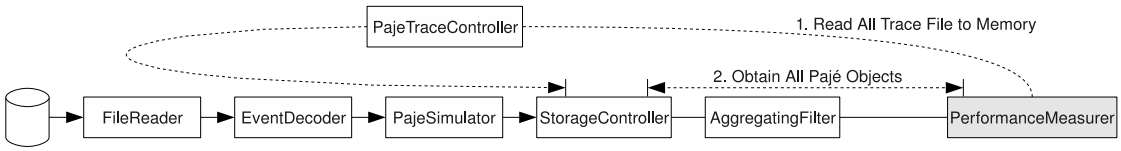


Figure 5.2 – Organization of Performance Tests developed with Pajé components.

We decided to remove the filters that depend on user interactions, since we are measuring only the performance of the core Pajé components. Figure 5.2 shows the configurations without these filters. The only filter we left is the AggregatingFilter, in charge of the scalability of the answers to the queries by the performance measurement component, and that does not require configuration by the user.

The basic algorithm for the performance measurements is to read the whole trace file and, after its completion, navigate through all objects in the memory. The Figure 5.2 also illustrates both steps with the dashed lines. Time measurements of both steps have been taken and the memory utilization is obtained at the end of program execution, just before the release of all objects stored in memory.

Synthetic generated traces were used as input for the tool. Since Pajé's AggregatingFilter solves the scalability problem caused by the amount of data per container, the generated trace files vary in their number of containers. As mentioned in previous Section, containers can be used to represent processes, threads, so changing their number in different inputs is reasonable enough to evaluate the simulator. The hierarchy used in the trace files is flat, meaning that all containers defined in the trace file are children of the same root container node. The different

traces range from 10 containers to 9 millions containers. We stopped the tests at 9 millions because of memory limitations of the machine used to run the tests. The containers of these inputs also have one thousand events that change their state through time.

In order to execute the performance evaluation, we used the nodes of the cluster *xiru* of the Parallel and Distributed Processing Group of the Federal University of Rio Grande do Sul. Each node has 8 Intel Xeon E5310 (1.60 GHz) processors with 16 gigabytes of main memory. The number of executions for a given trace file depends on the size of the file. For smaller files, we executed at least 100 times, but for largest files, at least 10 times. For all measurements for a given trace file, we removed 20% of the results (the 10% best and the 10% worst results) to keep the obtained results within a confidence interval. The remaining 80% of the results are used to create the average value, and then analyzed.

Figure 5.3 shows the results we obtained with the execution. The left graph depicts the execution time for both steps (step 1: Read and step 2: processing) of Figure 5.2. The x-axis of this graph shows the number of millions of containers, ranging from 1 to 15 millions. The y-axis is the time in seconds. The points indicate the measured values, up to 9 millions containers. The lines depict the linear regression technique generated with the measured points. We can clearly see that the evolution of execution times are linear, with the read step being more costly in terms of time than the processing.

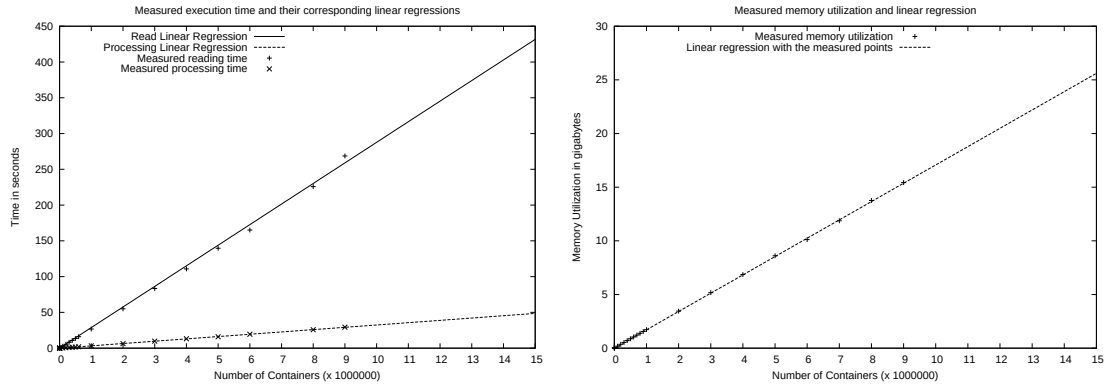


Figure 5.3 – Execution time and memory utilization obtained during performance experiments with Pajé.

The right graph of Figure 5.3 shows the memory utilization for the same experiment. Horizontal dimension indicates the number of millions of containers and the vertical dimension is the memory utilization in gigabytes. Points are measured and the line indicates the linear regression defined with the measured points. We can also observe a linear behavior in memory utilization required by the Pajé components.

5.1.3 Analyzing Pajé's Adoption

The advantages of using Pajé come from the software reuse, the scalability of the tool and the fact that Pajé deals with generic objects. The software reuse enables a fast development of additional components, the scalability has been shown through the performance evaluation tests

presented in the previous Section (results in Figure 5.3). We have been able to see that Pajé has a linear behavior in response times to queries and also in memory utilization. In the tests, we extrapolated the number of containers to see if Pajé can handle bigger quantities of containers in reasonable time. For one million containers, Pajé can read the trace file in about 25 seconds and return the data to the visualization components in about 3 seconds. Considering that each container is a process of a parallel application, we can argue that Pajé can manage trace files of parallel applications with one million processes in reasonable time.

The disadvantages of Pajé's adoption could be that a specific language and environment must be adapted to reuse its components. Furthermore, in terms of implementation, the tool that uses the components of Pajé must also have a GNUstep loop. Depending on which development environment is used, this means that another tool based on Pajé components must have at least two internal loops that must work together.

Considering advantages and disadvantages, we decided to adopt Pajé's components in the Triva prototype. The main reason behind this adoption is the possibility to handle generic objects, allowing the Triva implementation to be also generic, and the fact that Pajé is highly scalable. Next Section starts the Triva prototype description.

5.2 Triva Prototype Architecture and Overview

Figure 5.4 depicts the overall organization of the prototype, composed of modules that transform the trace data into Pajé objects, and then into the two types of visualizations: the 3D and the Time-Slice with treemaps. Because of the use of generic objects, the only trace-dependent part of the prototype is the one represented on the left of the Figure, denoted mainly by the DIMVisual Integrator and its sub-components specific to particular trace file formats. The white rectangles are existing libraries and tools that were re-used with minor adaptations; gray rectangles were implemented to be part of Triva prototype. This convention is used through the rest of this Chapter.

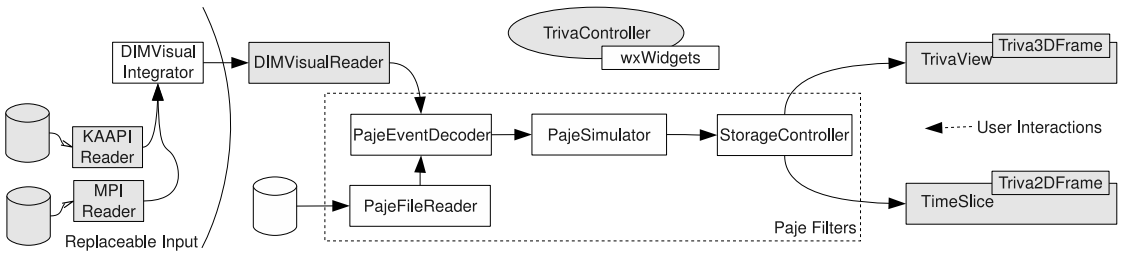


Figure 5.4 – Triva Architecture and Implementation Layout.

The TrivaController, written in C++ language, is in charge of the initialization of all the components and connecting them following the architecture presented in Figure 5.4. It also presents to the user a graphical interface, created using the wxWidgets library, under the form of a main window, with configuration options, menus and interaction mechanisms. The three dimensional scene and the treemap rendering is also initially configured and rendered.

The Pajé filters, represented by the dashed rectangle of the Figure 5.4, are the same as the ones used by the Pajé Visualization Tool [21]. Their implementation takes into account several issues like scalability and low response time to requests from the user interface. The first of the filters, PajeEventDecoder, handles the input generated by the DIMVisualReader and prepares it for the next module. The PajeSimulator transforms the events into visual objects. This transformation consists in the creation of a hierarchical structure of traces, using the basic types of Pajé. This structure, which represents the same information as in the trace files, is optimized for the visualization, and stored in the StorageController.

In the right most part of Figure 5.4, the interactions among the modules work in a two-way fashion. The interactions from right to left are the requests for new data. They are mostly triggered by user commands or changes in the configurations given as resource description. The interactions from left to right are the responses for the requests generated by the visualization.

To give a better description of the prototype, we split the explanation in three parts: one that details how the input is managed by the DIMVisualReader, another that explains the TrivaView and how the 3D visualization model is implemented, and the third named TimeSliceView, which explains the implementation of the second visualization model proposed in this thesis. Next sections detail these three parts in this order.

5.3 DIMVisualReader

The existing DIMVisual Integrator [70] is a software library to integrate traces from different data sources into a common format. As of today, the integrator is capable of generating a flow of events in the Pajé file format. The trace-dependent part of DIMVisual must be implemented to cope with specific formats. During this thesis, we implemented two trace-dependent modules: a KAAPI trace file reader and a MPI reader capable of reading traces generated by MPI applications.

Each sub-component of the DIMVisual Integrator is called a bundle, instantiated using the GNUstep library. A bundle means a self-contained binary object that can be dynamically loaded and linked during runtime within another program. After the initialization of Triva prototype, the user can configure the bundle it loads through the graphical interface. This interface acts through a configuration protocol, implemented in the DIMVisualReader module. Listing 5.2 shows the five methods of the protocol. The first three methods are used to check the bundles available, if a bundle with a certain name is already loaded and to load a specific bundle based on its name, respectively. The last two methods are used to configure a bundle that has been loaded. First, the function to get the configuration options is executed, returning a hierarchical structure with the options that must be defined to configure the bundle. These options are defined by the user through the graphical interface of the prototype. After this definition, the method setConfiguration is used to configure the bundle. A typical configuration holds information about trace files location in the file system, possible synchronization file and the kind of events that must be read by the module. Other options are also possible but are bundle-specific.

Listing 5.2 – Bundle Protocol Configuration.

```
- (NSArray *) dimvisualBundlesAvailable;
- (BOOL) isDIMVisualBundleLoaded: (NSString *) name;
```

```
- (BOOL) loadDIMVisualBundle: (NSString *) name;

- (NSDictionary *) getConfigurationOptionsFromDIMVisualBundle: (NSString *)name;
- (BOOL) setConfiguration: (NSDictionary *) conf forDIMVisualBundle: (NSString *) name;
```

Figure 5.5 depicts the behavior of the DIMVisualReader and related components. The DIMVisual Integrator generates as output a flow of timestamped objects that represents the application behavior. These objects are a high-level representation of traces, composed of Pajé events. The flow is received by the DIMVisualReader module, which implementation follows the internal protocol of Pajé [22]. The responsibility of the DIMVisualReader is to transform the flow of objects in textual representations using the Pajé file format. These representations are sent to the existing PajeEventDecoder filter and transformed to subsequent Pajé components. The DIMVisualReader does not send the objects directly to the PajeSimulator or the StorageController because the data generated by DIMVisual is different from the one used internally in Pajé.

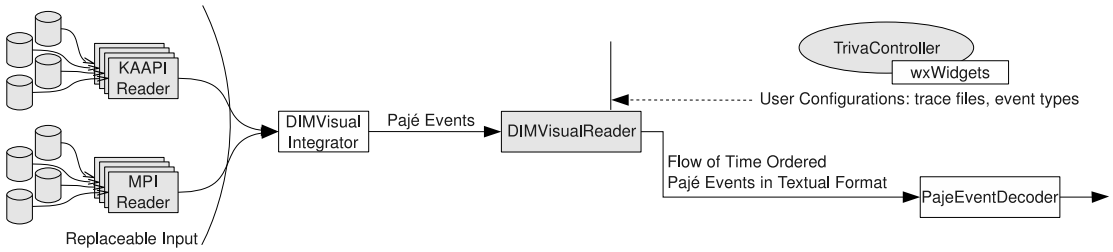


Figure 5.5 – DIMVisualReader Implementation and interactions with TrivaController.

The PajeEventDecoder is the first of the chain of re-used Pajé filters. The flow of textual events sent by the DIMVisualReader is received by this filter and transformed into a Pajé internal representation. As can be seen in Figure 5.4, the next filter in the chain of Pajé filters is the simulator. The simulator receives the decoded events and creates high-level objects based on the events. This high-level representation is basically an instantiation of the type hierarchy with timestamped objects, such as states, events and links. It is then stored in memory by the StorageController.

The main flow of information inside this part of the prototype comes from the trace files, depicted on the left part of Figures 5.5 and 5.4, to the Pajé filters, depicted in the dash rectangle of Figure 5.4. This flow of information, transformed in different ways by each component, stops in the StorageController. There, it is stored in memory and made available to the visualization parts of the Triva prototype. The flow is triggered periodically by the main loop of the prototype, handled by the TrivaController. More often than each half second, the controller sends a message to the DIMVisualReader to check if there is new data available. If this is the case, the new trace data is read and sent to the chain of filters up to the StorageController, where the flow of information stops.

Although the Triva prototype was mainly conceived to work with trace files, the implementation is also capable of handling events in an online fashion. For that, the DIMVisual Integrator must be attached to a source of events during the observation time of an application. Even if

possible, no tests have been performed to evaluate the online use of the prototype. The reason behind this decision is based on the amount of data generated in an online observation and the typical centralization of the analysis. We also intend with our approach to avoid the cost caused by the gathering and collection of data that is potentially distributed.

5.4 TrivaView

The 3D visualization model, presented in Chapter 3, is implemented in the Triva prototype through the TrivaView and related components. Figure 5.6 presents the overall organization of these components. The TrivaView module implements the Extractor part of the 3D model, retrieving from the flow of Pajé objects the containers and links, and redirecting the flow to the DrawManager component. The Entity Matcher part of the 3D model is implemented in three components of the prototype: TrivaApplicationGraph, TrivaResourcesGraph and TrivaTreemapSquarified. They receive the containers and links from TrivaView, and the resource description in files. The Visualization part of the 3D model, shown through the dashed line on the right of Figure 5.6, is implemented with four components: the Triva3DFrame, which holds the 3D scene, and the three managers that change this frame, the DrawManager, the AmbientManager and the CameraManager.

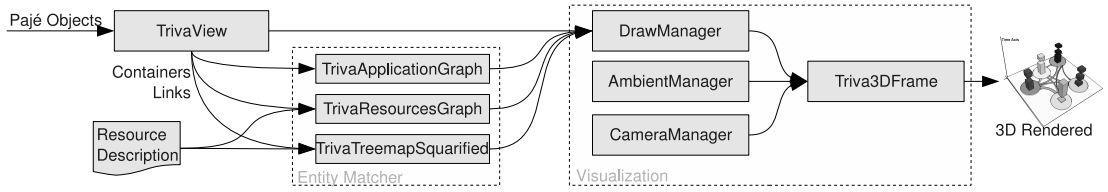


Figure 5.6 – TrivaView Implementation Layout

The details of the components related to the implementation of the 3D model are presented in next subsections. We start the description by presenting the two main libraries that are used in the implementation: the Ogre3D and GraphViz libraries. GraphViz is mainly used in the implementation of the visualization base, which description comes next with the algorithms and file format used as resources description. We end the details with the presentation of the 3D rendering scene.

5.4.1 External Libraries: Ogre3D and GraphViz

Two external libraries were used for the implementation of the 3D visualization model. The first one is called OGRE - Object-oriented Graphics Rendering Engine, which is a scene-oriented and flexible 3D rendering engine [43]. It is written in C++, designed to abstract the details of using libraries like OpenGL, and is released under the terms of GNU Lesser General Public License. Since Ogre3D is scene-oriented, it requires the creation of a hierarchical structure of scene nodes, attached to the Root Scene Node. Everything that is attached to this root node is supposed to be rendered.

When creating a scene, the scene nodes do not appear. The objective of scene node is to hold information about the position and scale in the 3D space. The objects that are rendered in the 3D space, such as cubes, cylinders, planes, and so on, must be attached to a scene node. All position and size operations that must be performed on a certain object should happen to a scene node in which this object is attached. Typical operations applied to scene nodes are rotations, translations, roll and pitch. If one of these operations is performed on a given scene node, all the objects that are attached to its descendants also receives the update. This hierarchical propagation of operations is especially useful since complex hierarchical structures can be changed by applying the operation to a single scene node. Besides, it is possible to remove one of these structures simply by removing the scene node that attaches it to the hierarchy headed by the root scene node. If the developer wants to make the structure visible again, it only has to attach it to the main hierarchy.

A scene is also composed by lights and camera. The Ogre rendering engine is able to manage ambient light and other types of lights, such as point, directional and spotlight. A scene must have at least one light to make objects appear, otherwise it is completely black. The developer must attach a camera to the scene in order to be able to observe in a computer what is rendered in the 3D scene. A camera is usually attached to a scene node where traditional position operations are performed. This way, the camera can rotate and move through the 3D space. The image that is usually seen in a computer screen window is what is visualized by the camera.

The second library used in the implementation of the 3D model is GraphViz [27, 34]. GraphViz is an open source graph visualization software. It gathers different graph drawing algorithms in the same tool. The basic usage of the tool is the generation of graphical images from the definition of graphs in a textual file format. Besides this traditional basic usage, GraphViz also works as a library that can be incorporated in other computer programs.

GraphViz, in its library form, is used extensively in the different base configurations of the 3D model, especially for the implementation of the application and the network/application graph combinations. The main functions of the library are *agnode*, to create a node, and *agedge*, to create an arc between two nodes. After the definition of the graph with these functions, the developer must call the function *gvLayout*, passing as parameter the name of the algorithm to position and render the graphical representation. At this moment, we can have access to several information regarding the graph, including for example the position of the nodes in a bidimensional space, the size of the nodes representation, the bezier curved lines that represented the arcs of the graph, and so on. It is this information that is used in the Triva prototype, especially the part related to the position information.

The GraphViz library is integrated in the prototype as described in the next Section, and the OGRE concepts are used in the description of the 3D rendering of the Triva prototype, in Section 5.4.3.

5.4.2 Base Configuration

Three types of base configuration were proposed in the 3D visualization model, back in Chapter 3. This Section explains how they were implemented, using as input the visual objects selected by the TrivaView module. Among the three visualizations, two of them must use graphs in their implementations: the application communication pattern and the combination of the

network topology and communication pattern. We use the GraphViz library to implement them. The other base configuration that consists in the treemap algorithm has been implemented from scratch.

Graph of the Application Communication Pattern

The application communication pattern, represented in Figure 5.6 by the component TrivaApplicationGraph, receives as input two types of Pajé objects: containers and links. As previously discussed in Section 5.1.1, containers may represent processes, threads, machines and so on, while a link is used to represent an interaction between two containers. For this part of the implementation, the relevant information present in container and link objects is the container identifiers. A container object has one identifier; and a link has two containers identifiers, one for the sender and another for the receiver. The algorithm that creates the graph using the GraphViz library is composed by two functions: *updateGraphData()* and *updateGraphLayout()*. Their simplified behavior are shown in listing 5.3.

Listing 5.3 – Algorithm to create the Application Communication Pattern based on containers and links.

```
graph_t *updateGraphData (graph_t *graph, list containers, list links)
        for container in containers
                agnode (graph, container.identifier);
        for link in links
                agedge (graph, link.send_identifier, link.recv_identifier);
        return graph;

GVC_t *updateGraphLayout (GVC_t *layout, graph_t *graph, string algorithm)
        gvFreeLayout (layout);
        gvLayout (layout, graph, algorithm);
        return layout;
```

The component responsible for the algorithm to create the communication pattern does not control how many information arrives. It is the responsibility of the TrivaView, in its controller form, to consider specific time intervals based on user choices. This means that if the user wants to see the communication pattern of the application occurring in a given time interval, the TrivaView must reset the graph already created by the TrivaApplicationGraph component and send it only containers and links present in that time frame. This has been implemented in the prototype by letting the user choose which time frame to analyze.

The function *updateGraphLayout()*, shown in Figure 5.3, defines the graphical layout of the graph. After calling GraphViz's *gvLayout()* function, there is enough information available to actually draw an image file with the visual representation of the graph. Among all this information, the Triva prototype uses only the bi-dimensional position of each node and the list of the arcs among them. So, after executing the function to update the graph layout based on the nodes and edges, the TrivaApplicationGraph sends the bi-dimensional position (x,y) of each container to the DrawManager. This manager is responsible for creating and positioning the visual objects that represent the graph in the visualization base of the 3D scene.

The user can also customize the layout by choosing which GraphViz's algorithm will be used to define the positions. As of today, there are five options: *dot*, *neato*, *fdp*, *twopi*, *circo*.

These options are extensively documented in the “Drawing graphs with GraphViz” documentation [33].

Graph of the Network Topology

The second type of base configuration is the mixing of the network topology and the application communication pattern. The implementation of this configuration is done in the TrivaResourcesGraph component. It is based on the resource description file provided to the component, as shown in Figure 5.6, and containers and links selected by the TrivaView component.

The resource description file matches dot's GraphViz format [27]. An example of such file is shown in listing 5.4, below. This simple example shows a list of machines that are interconnected by a switch. The component receives a configuration file like this and use the GraphViz's layout function to define the position of each node in the visualization base. As in the previous base configuration, only the bi-dimensional data defined by one the GraphViz's algorithm is used and passed along to the DrawManager component.

Listing 5.4 – Example of resources description showing the network topology, used to configure the TrivaResourcesGraph component.

```
graph G {
        "xiru-0.portoalegre.grenoble.grid5000.fr" -- "switch";
        "xiru-1.portoalegre.grenoble.grid5000.fr" -- "switch";
        "xiru-2.portoalegre.grenoble.grid5000.fr" -- "switch";
        "xiru-3.portoalegre.grenoble.grid5000.fr" -- "switch";
        "xiru-4.portoalegre.grenoble.grid5000.fr" -- "switch";
        "xiru-5.portoalegre.grenoble.grid5000.fr" -- "switch";
        "xiru-6.portoalegre.grenoble.grid5000.fr" -- "switch";
        "xiru-7.portoalegre.grenoble.grid5000.fr" -- "switch";
        "xiru-8.portoalegre.grenoble.grid5000.fr" -- "switch";
        "xiru-9.portoalegre.grenoble.grid5000.fr" -- "switch";
}
```

The second input given to the TrivaResourcesGraph is made of the containers and links, that come from the parallel application trace file. Since the component is pre-configured with the resource description file, the objective is to act upon the reception of containers by searching on which node of the network topology each container from the application trace should be placed. It is necessary to associate to each container from the trace to a location on the network topology, enabling the simultaneous analysis of both information.

There are several limitations to provide a successful association of containers from the trace to the nodes in the network topology. Usually, the only information present in the resources description file is the name of the machine. To provide a successful association with containers from the application trace files, the containers must hold some kind of location data. This data must come from trace events, registered by the monitoring system. In the Triva prototype, we used traces from KAAPI and MPI applications. For KAAPI, there are events that register the name of the machine where processes execute. Our tracing mechanism for MPI applications also registers the name of the machines involved in the execution.

When the association of containers to nodes of the network topology is successful, the TrivaResourcesGraph component sends to the DrawManager the position in the base for every node of the network topology and the position of every container inside a given node. By

doing this, the DrawManager has all the information necessary to place the visual objects in the visualization base of the 3D scene. The position of application containers inside a node of the network topology is also defined by a graph and implemented inside the TrivaResourcesGraph component.

Logical representation using Treemaps

The third base configuration is a logical representation of the resources using treemaps. For the Triva implementation, we decided to use the squarified version of treemaps [13], since it provides a better width/height ratio in the nodes representation. We implemented it in the component named TrivaSquarifiedTreemap, receiving as input two types of data: a resource description file and the containers of the application trace.

The format used for the resource description file that has to be provided to the component is in the Property List Format [3]. Figure 5.5 shows an example of this file. The example defines a hierarchical organization of machines, that are grouped by cluster, then by site which composes a grid. For each node of the hierarchy in the description file, there must be an attribute named type that indicates the type of the node on that level.

Listing 5.5 – Example of resources description showing the logical organization of resources, used to configure the TrivaTreemapSquarified component.

```
{
        name = Grid5000;
        type = grid;
        children = (
                {
                name = portoalegre;
                type = site;
                children = (
                        {
                        name = xiru;
                        type = cluster;
                        children = (
                                xiru-0.portoalegre.grenoble.grid5000.fr,
                                xiru-1.portoalegre.grenoble.grid5000.fr,
                                xiru-2.portoalegre.grenoble.grid5000.fr,
                                xiru-3.portoalegre.grenoble.grid5000.fr,
                                xiru-4.portoalegre.grenoble.grid5000.fr,
                                xiru-5.portoalegre.grenoble.grid5000.fr,
                                xiru-6.portoalegre.grenoble.grid5000.fr,
                                xiru-7.portoalegre.grenoble.grid5000.fr,
                                xiru-8.portoalegre.grenoble.grid5000.fr,
                                xiru-9.portoalegre.grenoble.grid5000.fr,
                                );
                        },
                );
                },
        );
}
```

The treemap algorithm is a space-filling technique that occupies all the space available for its drawing. The user defines, through the prototype graphical interface, the area in the visualization base that will be used to render the treemap. This information is passed to the algorithm implementation which starts a top-down and recursive traversal through the input hierarchy that

came from the description file. After the execution, all the nodes have their rectangles and their position defined in the bi-dimensional space of the visualization base.

The other type of input for the component is composed of containers from the parallel application trace. This second input is necessary because the TrivaSquarifiedTreemap must also define the position in the visualization base for every container of the application trace. This information will be used later by the DrawManager to place the containers on top of the areas reserved for a certain machines. The same association between resource and application container, present in the previous visualization base configuration, must be made here.

We have also implemented in the prototype the possibility of relating the size of each rectangle that represents a machine on the visualization base with the trace characteristics. This calculation is made depending on the options that the user chooses. Up to now, it is possible to use the number of containers in a given machine, the count of a specific states that appear in containers, and the amount of time of a given state in a container. After defining which metric will be used as squares size in the visualization base, the values for the leaf nodes of the hierarchy are defined and the treemap is computed. This can be performed at any time during an analysis.

As output, the TrivaSquarifiedTreemap send to the DrawManager the computed treemap data structure, that contains the position of each node and container.

5.4.3 Rendering the 3D Scene

The rendering of the 3D scene is controlled by three different managers: AmbientManager, CameraManager and DrawManager. The AmbientManager is responsible for creating the initial static drawings of the 3D scene and to manage the dynamic time scale rendered in the vertical axis. The static drawings do not change during the visualization of a trace file, but the timescale changes depending on interaction with the user. Figure 5.7 shows the scene nodes and entities organization created by the manager. The black circles represent scene nodes and gray squares represent entities that appear in the 3D scene. The static part is on the left of the vertical dashed line, and is composed of the Origin, and the three axis scene nodes, the ground plane and the three lines to show the three dimensions in the scene.

The dynamic time scale managed by the AmbientManager is depicted on the right of Figure 5.7, with N scene nodes and the same number of textual entities to indicate the timestamps that are rendered along the vertical axis of the scene. Whenever the time scale is changed by the user, all the objects on the right are freed and a new scale drawing is placed. The scene nodes of the time scale are attached to the YAxis scene node, but they are placed in the vertical axis according to the time scale currently in use.

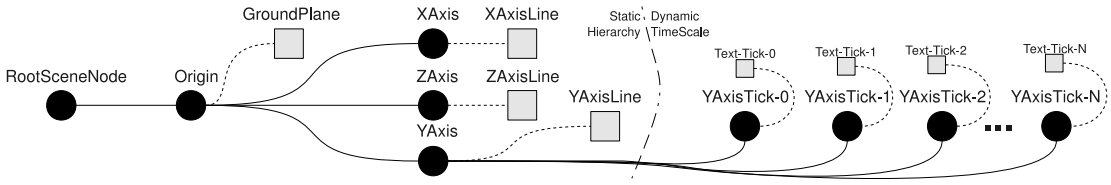


Figure 5.7 – Ogre3D's scene node and entities created by the AmbientManager to maintain the static part of the 3D scene and the time scale.

The CameraManager is another component that helps to manage the 3D scene. Its responsibility is to create and track the camera entity. Figure 5.8 depicts the Ogre3D's components used to manage the camera: there is a CameraNode, child of the root scene node, and two entities attached to it, the camera itself and a light that always point to the direction where the camera is looking at.

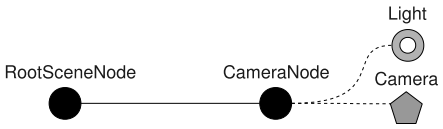


Figure 5.8 – Ogre3D's scene node created by the CameraManager to keep the camera entity of the 3D scene.

Configured by the TrivaController, the CameraManager also tracks the mouse and some keys of the keyboard to move the camera through the 3D scene. The implementation receives as input the arrow keys of the keyboard and transforms them into operations that are applied to the CameraNode. Every time the user uses one of the arrows, the prototype calculates a vector to move the camera. This vector is then applied to the CameraNode through a translation operation that also considers the orientation of the node in the 3D space. The manager also tracks the moves made by the user with the mouse. Based on them, the prototype determines two angles, one relative to the X plane and another relative to the Y plane, to be applied to the camera node through the operations yaw and pitch, respectively. This allows the camera to point to other directions based on mouse movements.

Rendering and Placement of the Visualization Base

The DrawManager is the main component that renders the 3D scene. It receives as input the configuration of the base already calculated by previous components, the positions of the containers in the base, and the timestamped Pajé objects to be placed in the vertical dimension according to their containers. The DrawManager takes these inputs and start the creation of a hierarchical structure of Ogre3D's scene nodes and entities. This structure is then rendered by the Ogre3D library in the Triva3DFrame of the Triva prototype.

Figure 5.9 shows the hierarchical structure that is created by the DrawManager to place the objects in the visualization base according to the input. As in previous Figures, the black circles indicate scene nodes, and the gray squares mean entities. On the left of the Figure, there is the scene node CurrentVisu, child of the root scene node. The use of this scene node enables the possibility of drawing more than one trace visualization on the same 3D scene. At this time, the prototype has only one of such node. The CurrentVisu scene node has two children: the ContainerPosition and the VisualizationBase. As the name indicates, the container position scene node contains a list of scene nodes ($C1$, $C2$, ...) that holds the position in the base of each container that comes as input to the DrawManager component. Each of these scene nodes has a sub-hierarchy composed by the visual representation and a 3D text (Draw and Text scene nodes). Each container scene node is used latter when the timestamped objects are attached to the scene. The other child of the CurrentVisu is the VisualizationBase scene node. It keeps the structure

for the current visualization base. In the Figure, the ResourcesGraph and the SquarifiedTreemap structures are depicted. The first one is the structure used for the drawing of the network topology and application graph. The second is the one that shows the treemap as base.

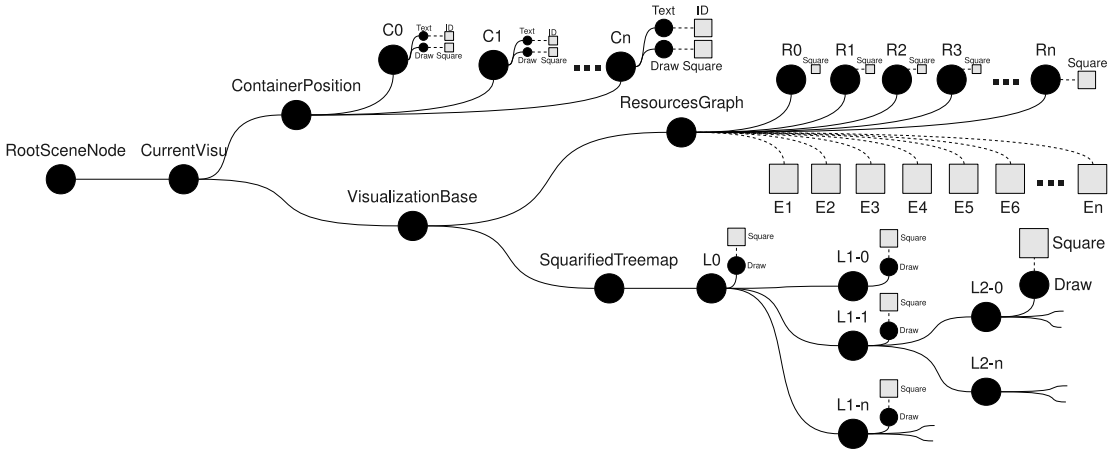


Figure 5.9 – Ogre3D's scene nodes created by the DrawManager to render the 3D scene for the visualization model.

The ResourcesGraph of Figure 5.9 maintains a dynamic list of scene nodes to represent the resources ($R0$, $R1$, ...). The resources are the ones sent by the TrivaResourcesGraph component as part of the network topology read from the configuration file in GraphViz's format. Besides the information about the resources, there is also a list of edges ($E1$, $E2$, ...) that are children from the ResourcesGraph scene node and represent the lines that interconnect the resources's square in the base. The scene nodes $R0$ ... Rn are positioned in the bi-dimensional visualization base according to the positions calculated by the TrivaResourcesGraph component. The position of the edges are then calculated based on who they connect.

The SquarifiedTreemap scene node of Figure 5.9 and its sub-hierarchy ($L0$, $L1-0$, $L1-1$, $L1-2$, ...) are created dynamically based on the configuration sent by the TrivaSquarifiedTreemap component of the Triva prototype. The sub-hierarchy reflects the hierarchy that comes from the logical organization of the configuration file. Each scene node has a square drawing attached to an auxiliary scene node to maintain scale and positioning.

Besides the two types for base configuration already described, there is also the application communication graph. This configuration, generated by the TrivaApplicationGraph component, is always present in the visualization. The scene nodes, the lines and possible arrows of its representation remain attached to the CurrentVisu scene node directly. This attachment can be controlled through the graphical interface, allowing the user to enable or disable to appearance of the communication graph of the application being analyzed.

As stated earlier, each Ogre3D scene node must have a defined position in the 3D space. This position is represented using the 3 coordinates: x, z and y. In Figure 5.9, all the scene nodes (the black circles) have the y coordinate set to zero. This places all scene nodes on the visualization base, as defined in the 3D visualization model Chapter. The other two coordinates (x and z) of all scene nodes of Figure 5.9 are defined by one of the three components that implement the

entity matcher (TrivaApplicationGraph, TrivaResourcesGraph and TrivaSquarifiedTreemap).

Rendering Timestamped Pajé Objects

The DrawManager also receives as input a flow of timestamped Pajé objects to be rendered in the 3D scene. In Section 5.1.1, we detailed that time-related objects are states, links, variables and singular events. Among these objects, we implemented only the 3D representation for states and links. These two types of objects can describe the behavior of several types of applications, since they can represent the execution of a function or a piece of code and also interactions among application's components.

Figure 5.10 shows the structure made by the DrawManager when drawing states and links into a 3D scene. The states are attached to the scene nodes of containers (from $C0$ to Cn). In the example of the Figure, each container holds n states, from $S0, C0$ to $Sn, C0$. The main reason for attaching the states to the containers scene nodes is that by doing so the states are placed exactly on top of the representation of containers in the visualization base. The only position information that must be computed by the DrawManager is the vertical position in the time axis. This computation for each state allows the correct placement of a visual representation of the state. This representation is a cube, and the color of the cube is associated to the value for that state. By doing this, all states of the same type will have the same color, facilitating their identification. The color scheme in fact is the same as the one used in traditional space-time visualizations.

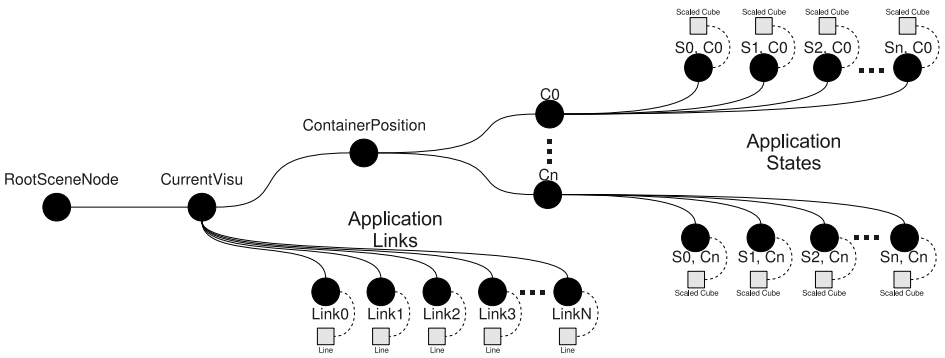


Figure 5.10 – Ogre3D's scene nodes created by the DrawManager to render the timestamped Pajé objects in the 3D scene.

The second type of timestamped-objects rendered is the links. When received by the DrawManager, links are transformed into a scene node that is attached to the CurrentVisu scene node. Figure 5.10 shows an example for that with the links $Link0$ to $LinkN$ scene nodes. Each link scene node has also a visual representation that is a line. The position of this line in the base dimensions are calculated based on the origin and destination of the links. For that, the DrawManager component obtains the x and z position of the containers involved (since a link is always between two containers) and creates the line between these two points in the base. After this, the DrawManager attributes the y coordinate of the beginning and end of the line by using

its two timestamps: one that indicate the beginning of the link and another the end. With the three dimensions defined for each extremity of the line, it is finally rendered in the 3D scene.

5.5 TimeSliceView

Previous Section described all the aspects of the implementation of the 3D visualization model. Most of these aspects are related to the TrivaView prototype component. Now, we present the implementation of the visual aggregation model proposed in this thesis. The main component of this implementation is the TimeSliceView, as can be seen in Figure 5.11. Another component of this part of the Triva prototype is Triva2DFrame, which responsibility is to draw the treemap in the visualization window of the prototype.

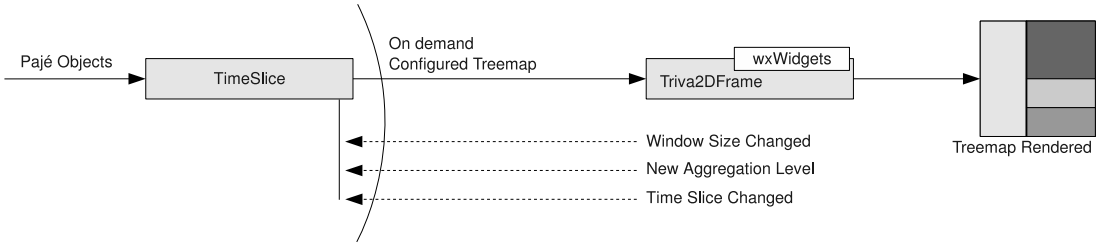


Figure 5.11 – TimeSliceView Implementation Layout with Notifications.

Figure 5.11 also details the interactions and notifications that happen during the TimeSliceView execution. The arrival of objects from the Pajé simulator (see Figure 5.4 for details) is depicted on the left of the Figure. The user interaction with the prototype can cause three different types of notifications that go from the Triva2DFrame to the TimeSlice: the change of the window size, a new aggregation level and the change of the time slice. All these notifications trigger the same chain of execution in the TimeSliceView component: creation of the behavior hierarchy, possible application of the aggregation operators and re-computation of the treemap. The resulting treemap configuration is sent as a response to the notifications and then rendered by the Triva2DFrame component.

Next Section presents the implementation that creates the behavior hierarchy. Afterwards, we present some information regarding the drawing procedures using the wxWidgets library functions.

5.5.1 Creating the Hierarchy

The Pajé objects and the type hierarchy of a trace in the Pajé format were described in Section 5.1.1. We observed that there are five different types of objects: container, state, link, event and variable. Besides, there is also a definition of a type hierarchy for each trace file in the Pajé file format. This definition enables, for a given trace file, to say that a process of a parallel application is of type container, and its behavior is of type state, for instance.

Figure 5.11 shows the implementation layout of the TimeSliceView and related components. The TimeSlice component is responsible for creating the behavior hierarchy that will be

shown in the visualization window through the Triva2DFrame component. In order to calculate the behavior hierarchy, the TimeSlice uses a set of methods from the Pajé filter protocol. The listing 5.6 shows the five methods (in the Objective-C language) of the protocol used by the TimeSlice component. The first is used to navigate through the type hierarchy, mainly through the containers, returning an array of containers type that are children of another container type. The second method is just used to confirm if a type is a container type (it can be of another kind, such as state, link, etc.). The third method is used to retrieve the Pajé type of an instance (container, state, link, event or variable). The fourth method returns an enumerator for all containers of the given type inside the given container instance. The last method returns an enumerator of the entities of the given type inside the given container between two timestamps.

Listing 5.6 – The five methods of the Pajé protocol used by the TimeSlice component to create the behavior hierarchy.

```
- (NSArray *)containedTypesForContainerType:(PajeEntityType *)containerType;

- (BOOL)isContainerEntityType:(PajeEntityType *)entityType;

- (PajeEntityType *)entityTypeForEntity:(id<PajeEntity>)entity;

- (NSEnumerator *)enumeratorOfContainersTyped:(PajeEntityType *)entityType
                                  inContainer:(PajeContainer *)container;

- (NSEnumerator *)enumeratorOfEntitiesTyped:(PajeEntityType *)entityType
                                inContainer:(PajeContainer *)container
                                   fromTime:(NSDate *)start
                                     toTime:(NSDate *)end
                                minDuration:(double)minDuration;
```

The TimeSlice component creates the behavior hierarchy using the methods above. The containers become the nodes of the hierarchical structure. The values of leaf nodes are calculated based on instances of the state type. At this moment, the implementation does not handle links, events and variables. Listing 5.7 shows the implemented algorithm to create the behavior hierarchy. Each time the method *createBehaviorHierarchy* is called, a node in the hierarchy is created. After the recursive call we can notice in the listing, the created nodes are attached to the parent node. The last line of the listing is executed when no further recursion is necessary, since the container does not have any sub-container. Being a leaf node of the behavior hierarchy, the node must find itself a value by calling the *timeSliceAt* method informing the container and its type.

Listing 5.7 – The implementation that creates the hierarchical structure based on the containers of the trace file.

```
- createBehaviorHierarchy: containerInstance
        containerType = [filter entityTypeForEntity: containerInstance];
        listOfTypes = [filter containedTypesForContainerType: containerType
                                inContainer: containerInstance];
        foreach type in listOfTypes
                if [filter isContainerEntityType: type]
                        /* Recursive call to create sub-nodes */
                        listOfContainers = [filter enumeratorOfContainersTyped: type
                                inContainer: containerInstance];
                        foreach container in listOfContainers
```

```
                createBehaviorHierarchy: container
        else
                /* Call the Time-Slice implementation */
                timeSliceAt: containerInstance ofType: type
```

The implementation of the method *timeSliceAt* is detailed in listing 5.8. The method receives as parameter the container and the state type that must be used to compute the values. The enumerator method, as previously stated, returns all the instances of that state type for the period of time between *sliceStartTime* and *sliceEndTime*. After receiving the list of state instances, the algorithm iterates through each of them, adding its value for each possible state name. This happens in the last method of listing 5.8. For example, considering a process as a container with a state: this state may have different names in an execution (blocked, running, barrier and so on). The last method of the *timeSliceAt* implementation will attribute the value for each of these names that corresponds to the intersection of the time slice and the duration of the state. If multiple occurrences appear in the same slice of time, the values are accumulated.

After the execution of implementations listed in 5.7 and 5.8, the containers and states will be reflected in the hierarchy as nodes and leaves, respectively. The leaves, which are created based on state Pajé instances, have values associated to them. The next step in the algorithm is to define the values for the intermediary nodes. This is implemented with a bottom-up algorithm that define the values of a node based on a sum of the values of its children.

Listing 5.8 – The implementation that returns a value for a given containers based on the states instances for that container.

```
- timeSliceAt: containerInstance ofType: type
        listOfStates = [filter enumeratorOfEntitiesTyped: type
                        inContainer: containerInstance
                        fromTime: sliceStartTime
                        endTime: sliceEndTime]
        while state in listOfStates
                stateName = [state name]

                startTime = [state startTime]
                endTime = [state endTime]

                laterStart = [startTime laterDate: sliceStartTime]
                earlierEnd = [endTime earlierDate: sliceEndTime]

                addValue: [earlierEnd timeIntervalSinceDate: laterStart]
                        forName: stateName
```

The previous algorithms, one to create the hierarchical structure and the other to define the value for leaf nodes, are sufficient to apply the squarified treemap visualization. The result of these algorithms is a hierarchical organization of objects, following the object-oriented pattern. The squarified treemap algorithm is implemented in the class that defines this hierarchical organization. This implementation is called just before sending the result to the Triva2DFrame component, which finally renders the treemap in the window.

In Chapter 4, we also presented the aggregation algorithm that is applied to simplify the behavior hierarchy created by the Time-Slice algorithm. The aggregation model is also implemented inside the TrivaView component, through a method named *limitHierarchy* which receives as parameter the hierarchy to be simplified and the new depth of the tree. The imple-

mentation of this method is shown in listing 5.9. The method is basically divided in two parts, one that does the aggregation, and another to do the recursion in the hierarchical structure. The first part, where the aggregation takes place, is implemented by obtaining all the children for a given node, then removing these nodes from the original structure. The obtained nodes are summarized based on the similar attributes. For example, if an instance of machine container has multiple process containers as children, which in their turn have two types of states (each one with a value); the aggregation algorithm will sum all the values of the same state type, remove all the nodes process and create a new node that is child of machine with the resulting aggregated value.

Listing 5.9 – Recursive implementation of the visual aggregation technique, applied to simplify a hierarchical structure generated by the Time-Slice algorithm.

```
- limitHierarchy: hierarchyNode toDepth: depth
        if [hierarchyNode depth] == depth &&
                        [hierarchyNode depth] != [hierarchyNode maxDepth]
                /* Create a summary of the children at this depth */
                children = [hierarchyNode children];
                [hierarchyNode removeAllChildren]
                summary = [hierarchyNode summarize: children]

                /* Inserting summary nodes back to the tree */
                foreach sum in summary
                        [hierarchyNode addChild: sum]
        else
                /* recurse */
                foreach child in [hierarchyNode children]
                        [self limitHierarchy: child toDepth: depth]
```

In the implementation of the aggregation method, we used only the sum function to do the aggregation. This, however, can be easily changed in the implementation or even transformed in an option of the user. The possible operators for the aggregation can be any of the operators discussed in Section 4.3.1.

5.5.2 Drawing with the wxWidgets library

After the creation of the behavior hierarchy, in its original or aggregated form, the responsibility of the Triva2DFrame component is to actually draw the rectangles, lines and textual representations. As previously stated, the component receives from the TrivaView a hierarchical object-oriented structure composed of nodes with treemap information already defined, based on the values defined by the Time-Slice algorithm and the visual aggregation technique.

The Triva2DFrame receives as input this hierarchical structure and goes through it obtaining information during drawing procedures. Some functions from the wxWidgets library [75] are used to draw in the visualization window: *DrawRectangle*, *DrawLine* and *DrawText*. The first function is used to draw the rectangle that represents a given node of the hierarchy. The information passed as parameter to this function are the width, the height and the bi-dimensional position in the visualization window. The *DrawLine* function is used to draw the timeline in the bottom of the visualization window. It appears only when the user moves the mouse pointer close to the bottom region of the window. The *DrawText* function is used when the user click into a rectangle: additional information about what that rectangle represents is drawn.

5.6 Summary

This Chapter has presented the implementation of the two visualization models proposed in this thesis. The first one deals with the three-dimensional representation of application traces to help developers visualize program behavior together with resources organization. The second is about the visualization scalability problem through a technique called Time-Slice that describes the program behavior in a hierarchy for a given time interval. This second technique is complemented by an aggregation model that, combined with a treemap representation, achieves scalable visualizations.

The two techniques are implemented in the Triva prototype, which is composed of several existing libraries and tools, such as the Pajé, GraphViz, Ogre3D, wxWidgets and others. The first part of the Chapter evaluates the advantages and disadvantages of using some Pajé components, especially its simulator component. Through a set of performance experiments, we shown that the current implementation of the tool is scalable enough to most existing parallel applications.

The second part of the Chapter presents the Triva prototype architecture and its components. We present the implemented DIMVisualReader module, capable of attaching the DIMVisual into the Pajé components directly, without passing through a file in its file format. Then, we present the details of the implementation of the three-dimensional visualization model, giving special attention to the description of the base configuration and how the 3D rendering is implemented. We end the Chapter with the implementation description of the Time-Slice technique and the aggregation model.

The next Chapter presents the results obtained with the Triva prototype in different scenarios. The scenarios range from real experiments in the Grid'5000 platform to the use of synthetic traces to show the resulting visualizations obtained with the prototype.

Chapter 6

Results and Evaluation

The last Chapter has presented the Triva prototype. It implements the two visualization models proposed in the thesis: the three-dimensional and the visual aggregation model. The Chapter details the general architecture of the tool, the implementation of the components and the external libraries used to support the handling of graphs and the three-dimensional scene.

The current Chapter shows the results we obtained with the prototype through the visualization of different traces, some of them generated synthetically, and others obtained with real executions of applications in a distributed and parallel platform. The results are composed of the visualizations generated by the prototype when the traces are used as input. The main objective is to verify if the 3D visualizations enable a better understanding of the traces considering the network topology and if the treemap visualizations computed by the proposed models allow large-scale analysis. For that, the results are divided in two parts: one that shows the three-dimensional visualizations, with the representation of the network topology; and the other part is composed of treemap views, trying to solve the visualization scalability problem of program analysis. Before diving into the description of the results, we detail in next Section the different traces used as input to the prototype.

6.1 Traces Description

As previously described in Chapter 5, the prototype must receive as input a flow of events in the Pajé format. The flow of events can be generated by using the DIMVisualReader component, or a file containing all the events. The visualizations offered to the user are always the same, no matter which of these options are used to enter trace data in the prototype.

This Section explains how the traces used in the prototype were generated or collected. By generation we mean that a set of traces used in the validation of the tool were synthetically created. The synthetic traces are necessary to facilitate the analysis of the prototype and the visualizations it creates. An example to justify the use of synthetic traces is the complexity of finding real traces to large-scale situations. The generation of such traces that reflects the behavior of applications running in many thousands of nodes is only possible if a large amount of resources is available, which is not the case. For these reasons, we implemented two tools to generate synthetic traces. One of them generates large-scale traces for the visual aggregation model

implementation, and the other complex topologies for the three-dimensional visualization.

Other set of traces were collected during the execution of parallel applications in distributed and parallel platforms. KAAPI and MPI applications were used in this case, the former being executed in the french Grid'5000 platform and the later in a cluster of the Federal University of Rio Grande do Sul, in Brazil. MPI applications are used for the sake of demonstrating how the prototype can handle traces from different types of communication libraries.

We believe that these two types of traces – synthetic and collected – illustrate common problems that are faced by parallel application developers in different situations. Next subsections detail how these traces were obtained.

6.1.1 Synthetic Traces

Section 5.1.1 detailed that a Pajé trace file is composed by three sections: the header, the type hierarchy and the timestamped events. The header is the only static part of file, where events are defined with their particular fields. The type hierarchy defines the types that will be present – such as cluster, machine, processor, processes, functions – and the hierarchy among them. The type hierarchy must be followed through the rest of the file in the timestamped events region.

Large-Scale Hierarchies

The first synthetic trace generator tool was created targeting the visual aggregation model. The tool is written in the Python language and receives as parameter a hierarchical structure that configures the generation of the trace. Listing 6.1 shows an example of configuration file that is passed as parameter to the tool. The file is organized hierarchically to reflect the type hierarchy that is generated as output. Each level (eg, Site, Cluster, Machine and Processor) has an attribute *container* that indicates the number of instances of that type that must be created by the tool. In the example, the configuration tells the tool to create 5 different sites, each one with 3 clusters, each cluster with 100 machines and each machine being composed of 4 processors. The attributes *alias* and *name* are used by the tool to comply with a trace generation required by the Pajé format.

Listing 6.1 – Example of configuration file for the large-scale trace generation tool.

```
config = {
        'container': 5, 'name': "Site", 'alias': "S",
        'child': {
                'container': 3, 'name': "Cluster", 'alias': "C",
                'child': {
                        'container': 100, 'name': "Machine", 'alias': "M",
                        'child': {
                                'container': 4,
                                'name': "Processor",
                                'alias': "P",
                                'statealias': "S",
                                'statename': "State",
                        }
                }
        },
        'appduration': 20,
        'cosine-max-x-axis-value': 7.5,
```

```
}
```

Still on Figure 6.1, the last level of the structure – Processor in the example – receives additional configurations: *statealias* and *statename* indicating the presence of a state on the containers created in that level. The time duration of the synthetic trace is configured through the *appduration*. The parameter *cosine-max-x-axis-value* controls the distribution of state values for the instances of containers in the last level. Its value is used to configure the cosine function from the interval 0 to the configured value. The tool maps the containers instances of the last level to the x axis of the cosine to find the amount of time – in percentage from 0 to 1 in the y axis of the function – a given container stays in one of two possible states. The remaining percentage is used to set the amount of time to the other state.

The graph of Figure 6.1 is configured using the data of the example in listing 6.1. The graph is used to define the duration of each of the two states available for every leaf instance container of the hierarchy. Using the configurations, the cosine function varies within the x interval $[0, 7.5]$. There are 6000 processors (result of the multiplication of all *container* attributes value: $4 \times 100 \times 3 \times 5$). The Figure also shows the definition of duration for the two states for the container number 4000. The value of the corresponding x value in the lower scale is 5. The cosine of 5 is 0.28. Since the values of cosine vary between -1 and 1 in the y-axis, we consider that this value of 0.28 represents 64% of the interval $[-1, 1]$. So, this percentage is used to define the amount of time of the State-0 for the container 4000, which is 12.8 seconds considering the total application duration of 20 seconds. The rest (7.2 seconds) is left to the State-1 of container 4000.

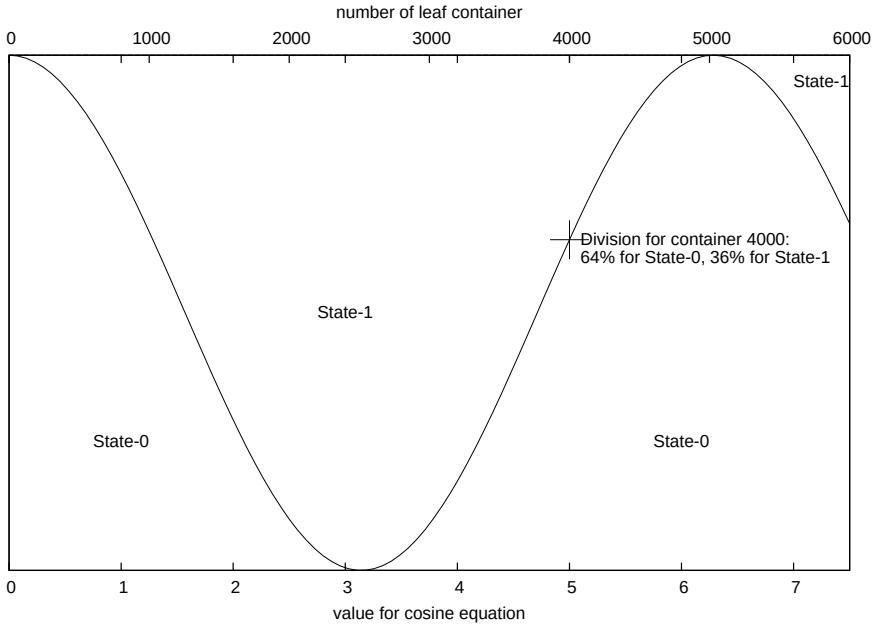


Figure 6.1 – State distribution among leaf containers using the cosine function.

Although the tool is implemented using the cosine function, it could be easily adapted to use other trigonometric functions. The way the state generation is implemented limit the study of different time intervals, as defined by the visual aggregation model. The positive side of the implementation is that it allows the fast generation of traces composed of hierarchies with thousands of nodes. The implemented trace generation tool takes less than 3 seconds to generate a hierarchy with more than 150 thousands leaf containers in a four-level hierarchy. A random state value generation was considered to implementation, but initial tests have shown that the execution time for large-scale hierarchies is too big when using a random number generator.

Typical Communication Patterns and Complex Topologies

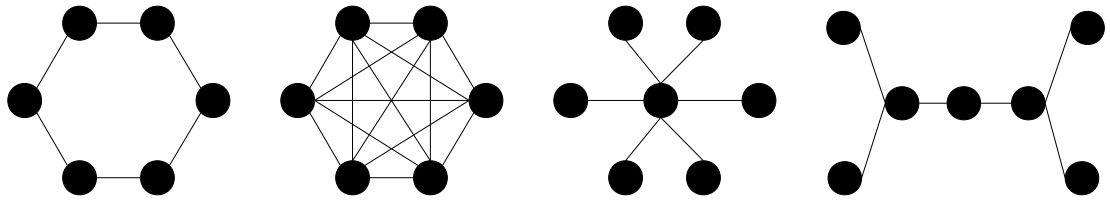
The second synthetic trace generator tool targets the three dimensional approach. The main objective is to generate trace files with traditional communication patterns, such as the ones used by master-slave or divide and conquer parallel applications. Listing 6.2 shows the configuration file used by this tool. It earns the basic configurations from the previous script, letting the user configure a hierarchical organization of containers if necessary. We implemented four types of communication patterns: ring, fully connected, star and hierarchical star. The user configures the type of pattern used through the option *apppattern*. If the user uses the hierarchical star communication pattern, an additional option called *nchildren* is necessary to configure the number of children in the communications. For example, if the *nchildren* parameter is set to 2, every container will communicate with other 2 containers. Each one of these two containers will communication with other 2 containers, and so on, forming a hierarchical communication pattern. The last options in the bottom of the listing are related to the Pajé links configuration. The options *linkalias* and *linkname* are used to configure the type hierarchy for the Pajé trace file, and the *linksource* and *linkdest* indicate which types of container can be used by these links.

Listing 6.2 – Example of configuration file for the synthetic communication pattern trace generation.

```
config = {
        # hierarchical definition section
        'container': 20, 'name': "Machine", 'alias': "M",
        'statealias': "E", 'statename': "State",
        'appduration': 20, 'cosine-max-x-axis-value': 7.5,
        #'child': {} # hierarchy with only one level in this example

        # communication patterns section
        'apppattern': "ring",  # ring, or full, or star, or hierarchical-star

        # parameters to "hierarchical-star" apppattern
        'nchildren': 2, #number of children per node

        # links configuration
        'linkalias': "P", 'linkname': "Link",
        'linksource': "M", 'linkdest': "M",
}
```

As previously stated, the four types of communication pattern that can be generated by the tool are the ring, the fully connected, the star and the hierarchical star. In the ring pattern, each container communicates exactly with other two containers, forming a single and continuous

pattern among all nodes. Figure 6.2(a) is an example of this pattern when there are 6 containers participating of the communications.



(a) Ring (b) Fully connected (c) Star (d) Hierarchical Star

Figure 6.2 – Different communication patterns generated by the second synthetic trace generation tool.

Figure 6.2(b) shows the fully-connected communication pattern, where all containers communicate with all other containers. Figure 6.2(c) shows the star pattern, where all nodes communicate with only one node. This type of pattern is typically found in master-slave parallel applications. The last communication pattern, represented in Figure 6.2(d), is a modified version of the star pattern, but with a hierarchical organization where each node has communications with other two nodes. In the example of Figure 6.2(d), the hierarchy is binary, but other configurations are also possible.

6.1.2 KAAPI Traces

KAAPI [35] stands for Kernel for Adaptative, Asynchronous Parallel and Interactive programming. It is a library that can be used by C++ developers to create parallel applications. The applications are composed of tasks and the data dependencies among them. In the beginning of the application execution, the KAAPI kernel spreads the tasks among the computing resources available. Afterward, during application execution, the kernel performs load balancing through work stealing algorithms.

Each KAAPI process executes the tasks defined by the programmer. When the tasks given to a certain process are finished, the process tries to "steal" the tasks from other processes of the application. The target process that suffers a steal is chosen randomly by the originating process. By doing this random steal, KAAPI guarantees good load balancing for the application at a small cost.

The KAAPI library is internally organized in levels. Common levels of the implementation include the generic kernel work stealing of threads (Kernel), data flow graph management (DFG), remote work stealing (WS), network (NET), static scheduling (ST) and the fault tolerant (FT) levels. Every level implements a sub-set of KAAPI functionalities. The FT level [11], for instance, is responsible for dealing with resources outage, such as the loss of processes and tasks during runtime.

Each level is instrumented in the implementation so its behavior can be traced during application runtime. In our work, we have used the events generated in the generic kernel (KERNEL) and work stealing (WS) levels. These events register the remote work stealing activities of

KAAPI library, such as the stealing attempts when a given process remains without any task to execute. Figure 6.3 shows the KAAPI events that are considered in our work and how their combination define the states of a KAAPI process. The events *Core_Idle_0* and *Core_Idle_1* are registered in the Kernel level and define the period on which a given process is not executing tasks defined by the programmer. The events *Core_Rsteal* and *Core_RetRsteal* define the moment where the KAAPI library is trying to steal a task from another process. Additional information if the steal was successful or not, and the target process, are also registered. All these events are registered by the *K-Processor* threads of the application, which are responsible for executing tasks during runtime.

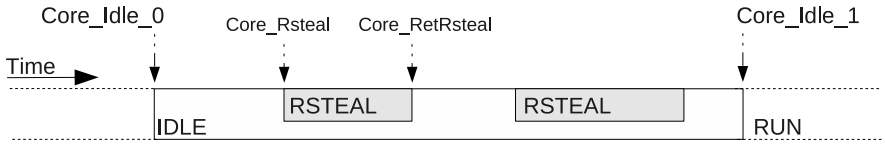


Figure 6.3 – KAAPI Events to monitor the remote work stealing activities of the library.

Still on Figure 6.3, the combination of the KAAPI events allows the definition of three possible states for a K-Processor: IDLE, RSTEAL and RUN. The IDLE state is defined as the time where the processor is not executing tasks. During the IDLE state, the K-Processor can execute a number of remote work stealing requests, which defines the RSTEAL state. The RUN state is defined by the period where a given K-Processor is not in the IDLE state.

The traces of KAAPI applications have been obtained in the Grid'5000 platform. For every execution, the processes register on which machines they were executed, the beginning timestamp, and the global KAAPI identifier. This information is registered by another level of the KAAPI stack, named Util. The registered data is used to properly convert the information to Pajé traces after the execution.

Considering the Grid'5000 platform as execution environment, KAAPI registers the name of the machines used in an application execution. The name of the machines, as obtained by the Domain Name Server (DNS) of Grid'5000, allow the definition of a type hierarchy with the following levels: Grid, Site, Cluster, and Machine. All this information is obtained from the machine names. For instance, during a KAAPI execution, each process registers the name of the machine where it executes. Considering a Sophia-Antipolis machine with the name *azur-7.sophia.grid5000.fr*. From this name, it is possible to obtain the machine – *azur-7*, the cluster – *azur*, and the Grid'5000 site – *sophia*. The rest of the hierarchy is composed of the global KAAPI identifier and the instance of K-processor. Therefore, the resulting Pajé hierarchy for the KAAPI traces is the following: Grid, Site, Cluster, Machine, Process, K-Processor. The hierarchy is completed with the three possible states for a K-Processor (IDLE, RSTEAL and RUN).

The conversion of KAAPI traces to the Pajé file format happens with the help of DIMVisual. The input modules are able to read the KAAPI trace format and convert them to common Pajé events, such as *PajeSetState*, *PajePushState* and *PajePopState*, to handle the definition of the three states of the K-Processors. Other Pajé events, such as *PajeCreateContainer*, are used to create the containers of the type hierarchy of KAAPI traces.

6.1.3 MPI Traces

One of the main benefits of using the Pajé file format as input for the Triva prototype is related to the generic use of the tool. In order to show a different example, we used trace files generated during the execution of MPI [37] parallel applications. The different applications were the ones available in the NAS Parallel Benchmark (NPB) [8], which contains a number of applications to handle numerical aerodynamic simulations. Since the benchmark includes some applications developed in Fortran, we considered for the traces only the applications implemented with the MPI specification and in the C language.

The traces of NAS applications were obtained through the instrumentation of the Mpich library, using a wrapper for each MPI operation [31]. The wrapper can be enabled through the presence of the MPE – Multi-Processing Environment, when compiled together with the Mpich library. All MPI operations are registered using this instrumentation tool. Additional information in point-to-point and collective functions are also registered, such as the origin and destination of the messages. As of result of an execution, a single trace file in the Pajé format is created.

The top part of Figure 6.4 shows some events that are registered by the instrumentation. For every MPI function, the instrumentation registers the moment it began and when it returned. These events are transformed into the Pajé format mainly by using the *PajeSetState* event. The state *RUN* is used to indicate that no MPI function is currently in execution. Others states for MPI processes are directly mapped from the names of the MPI operations, giving, for instance, a state *MPI_BCAST* for a MPI_Bcast operation. The operations that are related to message-passing, such as point-to-point or collective operations also generate *PajeStartLink* and *PajeEndLink* events.

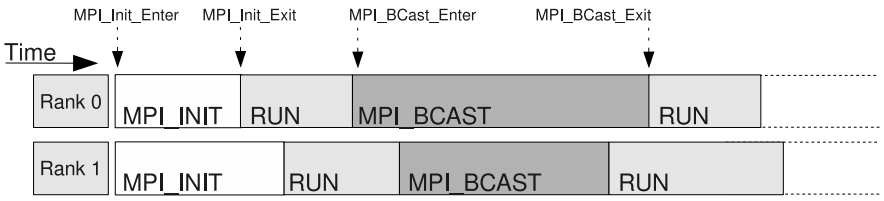


Figure 6.4 – Events registered during the execution of an MPI application (all MPI operations are registered).

Since the objective of the MPI traces is to show only that it is possible to handle this type of data in the prototype, only small-scale executions were performed in a cluster located in Porto Alegre, Brazil. The NAS benchmark executions used at most 16 machines of the cluster.

6.2 3D Visualizations

The 3D visualizations of the Triva prototype are created by the TrivaView (see Section 5.4 for details). This component manages the base configuration and the rendering of timestamped objects in the 3D space. This Section presents the 3D visualizations obtained with the use of

synthetic and real trace data. The main objective is to observe the capabilities of the 3D approach in the visual detection of communication patterns, and the mapping with the network topology.

We start with a general description of the 3D visualization generated by the prototype, in next sub-section. Then, we show the visualization of known communication patterns and finish the Section with the use of KAAPI traces and topological representations of Grid'5000 platform.

6.2.1 Description of the Visualization

The basic three dimensional visualization generated by the Triva prototype can be observed in Figure 6.5. It shows two processes, A and B, that interact with each other. Different tonalities of gray represent the possible states in which a process can remain through a period of time. In the Figure, the light gray represents the Blocked state, and the dark gray represents Executing. RGB Colors are extensively used in the prototype but were removed from the representations in this text. The communication between two processes is represented by a line connecting them.

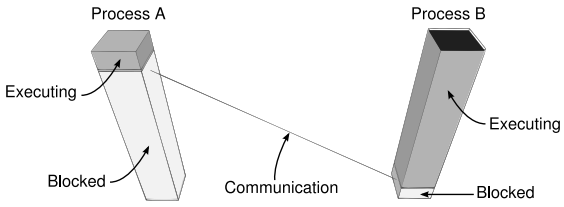


Figure 6.5 – Simple 3D visualization created by Triva with two processes.

Every state in the visualization can be clicked with the mouse to obtain more information about it. The related information includes the start and end timestamps for that state instance, which monitored entity it belongs and the name of the state. The lines that interconnect the processes can also be clicked to obtain more information.

Figure 6.6 shows another configuration on the visualization base. In the screenshot, we can notice the presence of two machines representation, X and Y, and a line to represent the interconnection among them. The application components, represented by processes from A to F, are placed according to the location in which they were executed. Processes A, B and C on top of machine X, the rest on top of machine Y. The lines interconnecting these processes represent the communications among the processes. In the example, there are inter and intra-machine communications.

When interacting with the visualization of Figure 6.6, the user is able to obtain information about every machine and the characteristics of the interconnection in the visualization base. This information is given to the user if it is available in the resource description file used to configure the prototype.

The Figure 6.7 depicts the visualization window of the Triva prototype. The graphical interface is managed with the help of the wxWidgets, including the menu, the status bar and the scrolling bar on the right. The 3D scene is rendered in the middle of the window, as depicted. All the messages towards the user, such as the information about a state, a process or a link, are shown through the status bar in the bottom part of the window. Through the menu, the user is

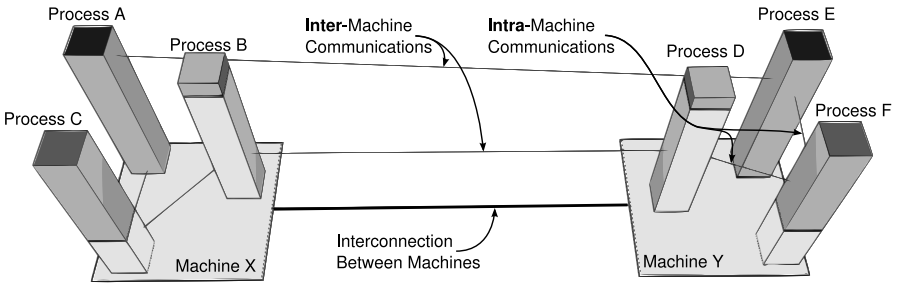


Figure 6.6 – Processes representation with network interconnection among two machines.

able to configure the visualization base, the time slice of the current analysis and the options regarding the movements of the camera inside the 3D space. The menu also enables the configuration of the trace files, through the customization of how KAAPI trace files will be read into the prototype.

The user 3D interactions are implemented directly in the 3D scene, through keyboard events or mouse movements. The user can, for instance, type the Ctrl key and the left mouse button to move one of the process representations in the visualization base. Other combinations of keys enable the selection of more than one process representation to move them together, and so on. Additional combinations can be easily implemented in the prototype.

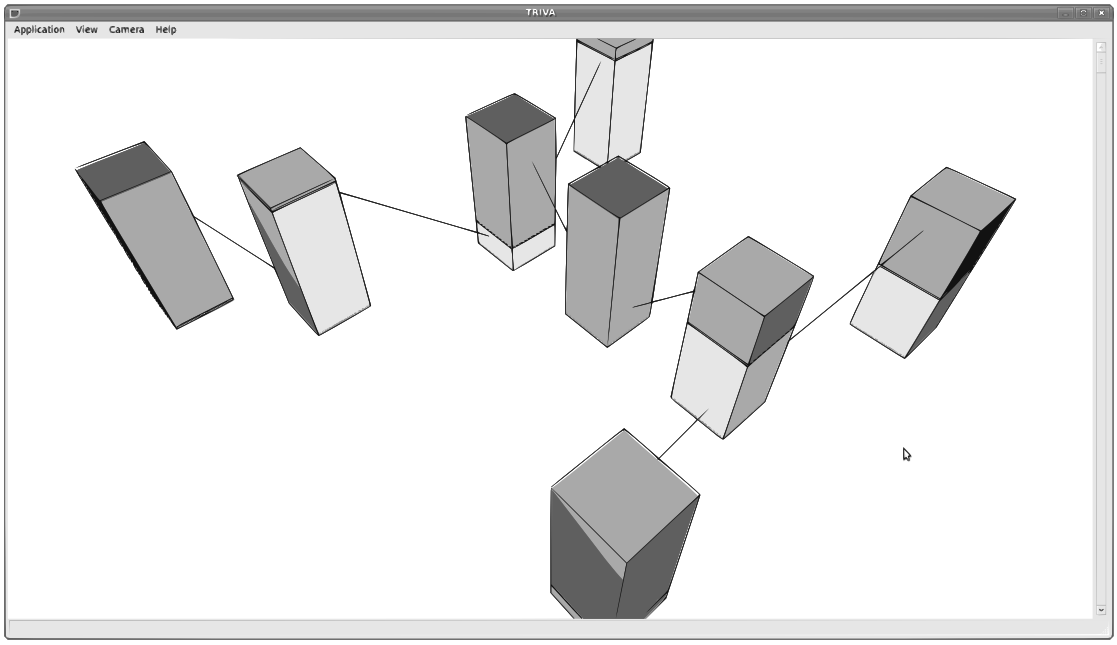


Figure 6.7 – The visualization window of the Triva prototype.

6.2.2 Communication Patterns Analysis

One of the first benefits obtained with the 3D approach is the observation of communication patterns. These patterns, when visualized through traditional space-time representations, are hard to analyze since only one dimension is available to depict the way processes interact among them. Using the synthetic trace generation tool, explained in previous Sections, we generated simple and known patterns. They include a ring, a fully-connected and a star communication pattern. Figure 6.8 depicts these three patterns, created using the Triva prototype with three different traces generated by the automatic trace generation tool.

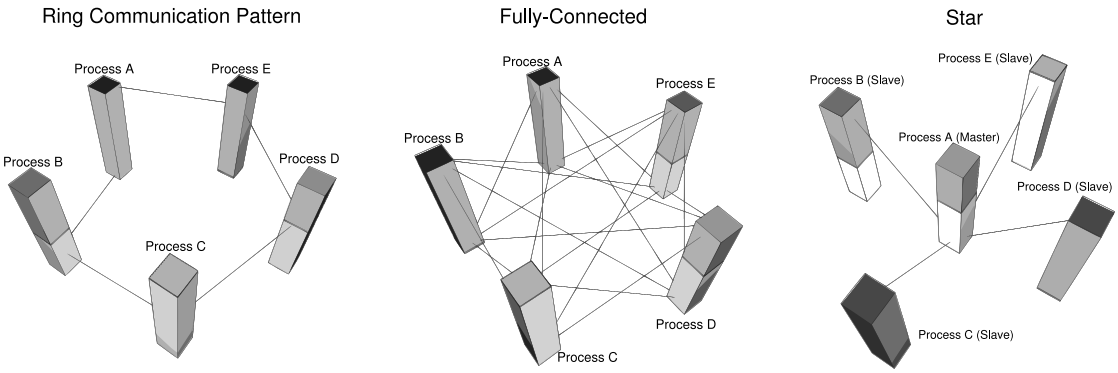


Figure 6.8 – A ring, a fully-connected, and a star communication pattern visualized with the Triva prototype.

The leftmost 3D view of Figure 6.8 shows a ring communication pattern, composed of five processes from A to E. The communication starts in the process A and goes through processes B, C, D, E and it finally comes back to the origin. We can observe in the vertical dimension that the beginning of a communication between the process D and E, happens after the reception of a communication in process D, indicating a sequential ring pattern. This identification, brought by the 3D approach, enables the user to see the difference in cases that the communication occurs in parallel. The center 3D view of Figure 6.8 shows a fully-connected communication pattern among the five processes. Observing the vertical axis, we can notice that the communications from one process to others starts in the beginning, close to the visualization base. The third communication pattern is on the rightmost part of the Figure, showing a star pattern with a central process. This pattern is commonly used in master-slave parallel applications. The star view shows an example of master-slave where process A is the master and the others, from B to E, are the slaves.

In order to compare the 3D with the traditional space-time visualization, we used Pajé to visualize the fully connected trace. The final 2D representation is shown on Figure 6.9 with five processes listed vertically, along with their states in the x-axis. Links are represented by the arrows. Comparing these views, we can notice some of the benefits of the 3D approach, where the communication pattern is more clearly observed.

The synthetic trace generator is also capable of generating a hierarchical star pattern. Using a trace generated with this tool in the Triva prototype, we obtain the visualization of Figure 6.10.

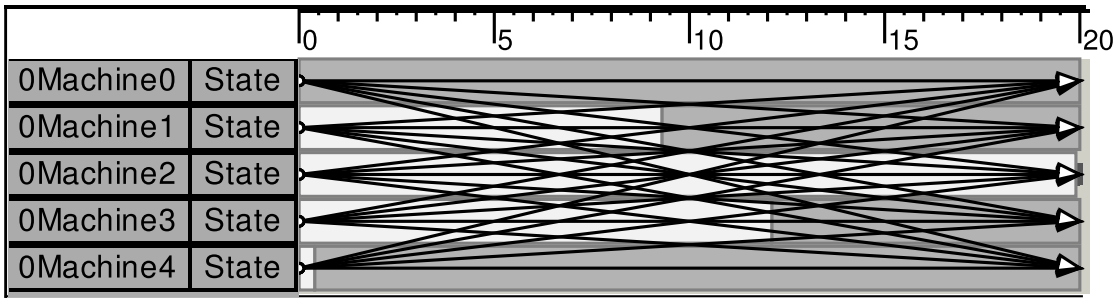


Figure 6.9 – A fully-connected communication pattern with five processes represented in the SpaceTimeView of Pajé.

The view shows seven processes with a first level master, the process A, that communicates with the second level masters, processes B and C. The others processes are connected to the second level masters and behave as slaves. This communication pattern can be observed in the beginning of applications built based on divide-and-conquer algorithms. They show in a first phase the divisions of work like a hierarchy.

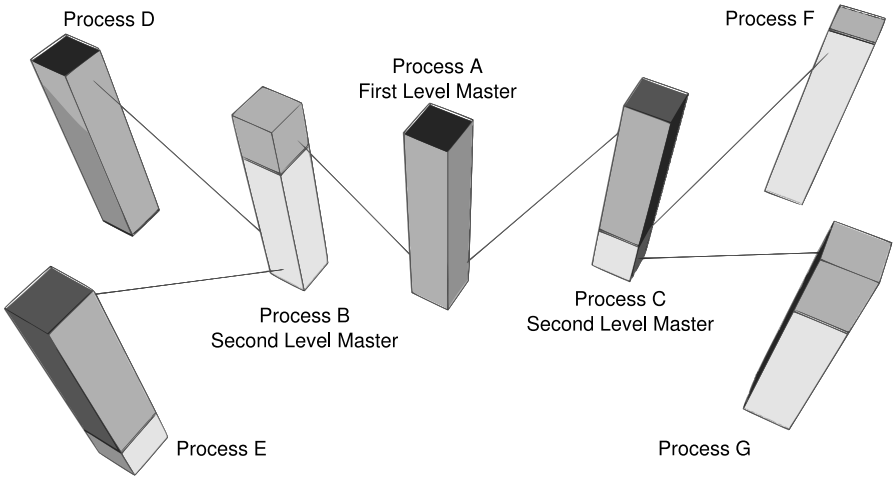


Figure 6.10 – A hierarchy star communication pattern, commonly used in divide-and-conquer algorithms, with a visualization of Triva.

The analysis of these communication patterns enables the observation of possible problems in the development of parallel applications. Suppose a developer decides to create a divide-and-conquer algorithm. After the implementation, the Triva prototype can be used to analyze if the communication pattern of the implementation is correct. The developer can also guess if a different number of levels could improve the performance of the algorithm, by analyzing the time a certain configuration take to execute. Another benefit of the Triva prototype is when the communication pattern of an application is unknown to the developer. In this case, the only thing to do is to execute the application once and visualize it in the prototype to understand the

possible patterns of the application under investigation. This is faster and easy to understand when compared to a traditional code analysis spread in several source files of the application (assuming it is even available).

6.2.3 KAAPI and the Grid'5000 Topology

This Section describes the results obtained with real application traces gathered from different experiments with KAAPI applications on the Grid'5000 platform. We selected six different scenarios to present these results, which consider as network interconnection the topology present in the Grid'5000.

Scenario A: 26 processes, two sites, two clusters

The first scenario is a KAAPI application composed of 26 processes. Each process is assigned to one distinct machine, resulting in an allocation of 26 machines. Half of them are allocated in the cluster *xiru*, at *portoalegre*, and the other half in the cluster *grelon*, at *nancy* site. Figure 6.11 depicts the 3D visualization generated by the Triva prototype of the application trace. The visualization base is configured to hold the network topology that interconnects both sites. In this example, we are using a hypothetical topology just to illustrate the analysis. The actual interconnection between *portoalegre* site and the rest of the Grid'5000 is a VPN, with several physical hops through the internet.

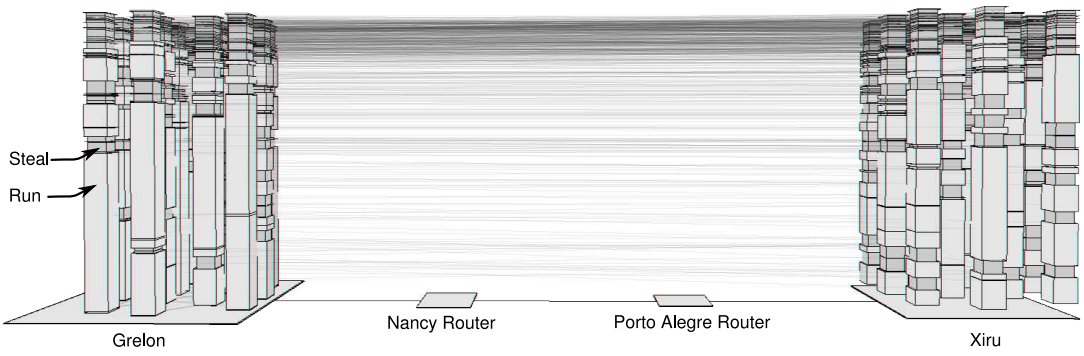


Figure 6.11 – A side-view generated by Triva with traces from 26 processes.

The first thing to be noticed on Figure 6.11 is the vertical bars representing the processes of the KAAPI application. The light gray represents the state *Run* and the dark gray represents the state *Steal* of a given process, as indicated in the leftmost part of the Figure. We can also observe in this Figure the horizontal lines connecting the processes from different sites. They represent the work stealing requests performed among the processes of the application. When the user is interacting with such visualization, it is possible to obtain information for every state and link represented. If a resource description with additional data about the interconnections is provided to the prototype, the user is capable to obtain such data through the visualization, by pointing the mouse to the squares and lines in the base. We can also notice in the Figure the distribution of steal requests in time.

Scenario B: 60 processes, two sites, three clusters

The second scenario is a KAAPI application composed of 60 processes, one per machine, that are executed in two sites of the Grid'5000. The site *nancy* contributes to the execution with 30 machines from the cluster *grelon*, at the same time that the site *rennes* has an allocation of 25 machines from cluster *paramount* and 5 machines from cluster *paraquad*. We consider in this case a topology where every site has its own router, where all clusters from that site are connected to. The routers of the two sites have a direct connection. Therefore, in this example when a message is sent from a cluster in one site to a cluster in other site, it has to go through the two sites routers.

Figure 6.12 shows two screenshots of the Triva Prototype generated during the visualization of the trace file for this scenario. The text and dashed lines were manually inserted to improve the understanding of the example. The image **A** of this Figure shows the total execution time with a small time scale, making all objects close to the visualization base. The dashed line on this image depicts the site separation between *rennes* with two clusters and *nancy*, with only one cluster. We can observe in this time scale that a large number of work stealing requests occur between *grelon* and *paraquad* clusters, mostly because of the higher number of processes executed on them. Analyzing these requests with the network topology, the Triva prototype allows the user to view that all the requests from these clusters must go through two routers of the interconnection. Such situation might lead to performance issues. A hierarchical work stealing is under investigation by the KAAPI team in order to overcome these problems.

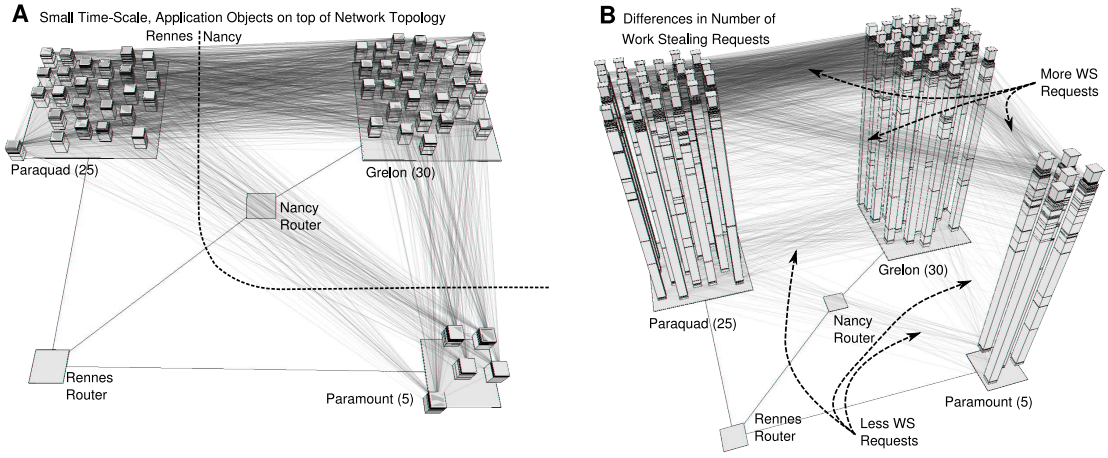


Figure 6.12 – Two screenshots of the prototype Triva during the visualization of an application composed of 60 processes, with different time scales.

The prototype also allows the dynamic change of the time scale, using the mouse wheel. The image **B** of Figure 6.12 shows the total execution time for the traces of this scenario, but with a larger time scale. Through this image, it is possible to see differences in the work stealing behavior in different intervals of time of the execution. It can be noticed that in the beginning there is less work stealing requests when compared to the end. It is during the end of the execution that less tasks are available for execution and processes start to try to steal more. This behavior

is expected considering the current implementation of KAAPI, where random steal requests are performed when processes are idle.

Scenario C: 100 processes, three sites, four clusters

The third scenario is an application composed by 100 processes, one per machine, allocated in four clusters that are in three different sites of Grid'5000. The allocation is as follows: cluster *grelon* with 30 machines at *nancy* site; *pastel* with 40 at *toulouse*; and 25 machines from *paramount* and 5 from *paraquad* at *rennes* site. The network interconnection here is constructed as in the previous example. In this scenario, we consider that the three routers are fully connected.

In previous scenarios, we observed screenshots where all the execution time is represented, sometimes with different time scales. The Figure 6.13 shows two screenshots where only a part of the execution time is drawn. This is possible in the prototype through an interactive configuration where the user specifies which time slice is rendered. The image **A** of the Figure shows the work stealing requests at the beginning of the application. The dashed lines separates the three different sites. As on previous cases, each cluster name has a number which indicates how many processes are executed on that cluster. We can clearly observe that in the beginning the number of stealing requests is considerably lower compared to the end of execution, shown on the image **B**.

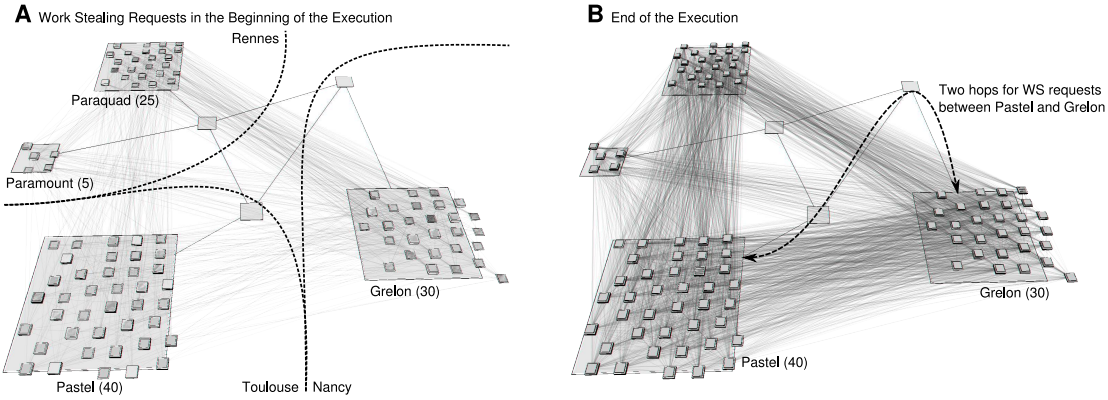


Figure 6.13 – Two visualizations with different time slices of an application composed of 100 processes.

The image **B** of Figure 6.13 also shows, through the dashed arrow, the path that all work stealing requests must follow from the cluster *pastel* to the cluster *grelon* and vice-versa. We can see with the rendering of the network topology that these requests must go through two routers in order to arrive in the destination. The visualization in this case may suggest that big cluster allocations for this particular execution should be placed in the same site, avoiding two hops for stealing requests. Small allocations could then be placed on other sites, because of the smaller number of steal requests generated by these small allocations.

Scenario D: 200 processes, 200 machines, two sites, five clusters

The KAAPI application of scenario D is composed of 200 processes, in 200 machines. The machine allocation is divided in two sites: *rennes* and *nancy*. The number of machines allocated in each site is equal, but inside each site the allocation differs in number of machines per cluster. The image **A** of Figure 6.14 shows the number of machines for each cluster allocated and also the network topology that interconnects the two sites. As in previous scenarios, the dashed line is used to separate the sites. In order to illustrate another benefit of our visualization, we consider for this scenario additional information regarding the network interconnection between the routers and three clusters. We consider here that the bandwidth available between *paravent* and *grillon* clusters, through the two routers, is of 100 megabits. The link between the *grelon* cluster and its router is of 1 megabit, as depicted in image **A** of the Figure.

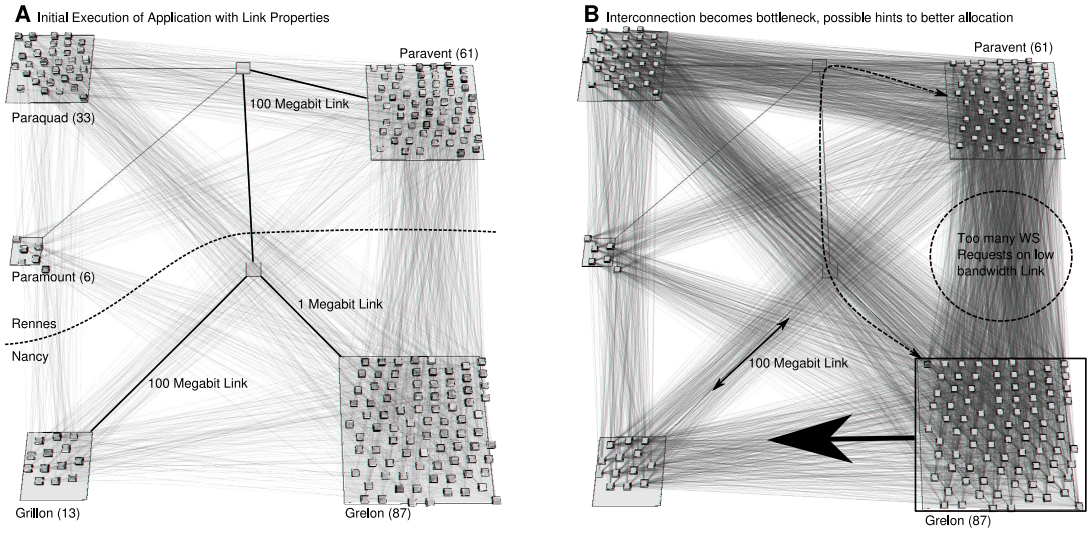


Figure 6.14 – Two top-views with a network topology annotated with bandwidth limitations, showing the benefits brought by the 3D approach.

In this scenario, there are 87 processes running on *grelon*, and 61 on *paravent* cluster. Let us consider only the work stealing requests between these two clusters, as depicted by the dashed circle of the right image of Figure 6.14. The dashed arrow of the same image indicates that these requests must pass through the 1 megabit link. The visualization suggests that a smaller number of processes should be placed in a cluster with such a slow bandwidth. If, for instance, the processes of cluster *grelon* were executed on cluster *grillon* instead, the execution could have a better performance.

Through the example of this scenario, we can notice the importance of analyzing the application performance together with a topological representation. If this type of visualization, such as the one present in image **B** of Figure 6.14, is not present, the analyst could obtain wrong conclusions about the performance of its application.

Scenario E: 648 processes, two sites, five clusters

The KAAPI library has a random work stealing mechanism. It means that whenever a process has no further tasks to process, it selects randomly another process to perform a stealing request. This random behavior is an easy and simple way to perform load balancing, being a distributed solution that scales well. The scenario E intends to show the resulting communication pattern caused by the KAAPI work stealing implementation in a large-scale situation with topological data. The network topology configuration is the same of scenario D, and the same number of machines is used to the execution of the application. The only difference here is that a higher number of processes is launched, resulting in 648 processes.

Figure 6.15 shows a screenshot of the Triva prototype when configured to show the behavior of all the execution time on top of the network topology. We can see the processes distribution among the clusters, which square size in the base is directly related to the number of processes in the cluster. Considering the five clusters of this execution and the random work stealing mechanism, it is expected to find steal requests from all clusters to all others. The four arrows, drawn manually on the view, put in evidence this behavior for the cluster *grelon*. We can see that other clusters also perform steal requests the same way, having as targets processes from all other clusters.

Scenario F: 2900 processes, four sites, thirteen clusters

The last scenario is an application composed of 2900 processes, executed in 310 machines that were allocated in clusters of four Grid'5000 sites. The machine allocation is as follows: 60 machines from *lille* site (41 - *chinqchint*, 10 - *chti*, 3 - *chuque*, 6 - *chicon*); 100 from *rennes* (61 - *paravent*, 6 - *paramount*, 33 - *paraquad*); 50 from *bordeaux* (5 - *bordereau*, 22 - *bordeplage*, 23 - *bordermer*); and 100 from *sophia* site (48 - *azur*, 42 - *sol*, 10 - *helios*). The objective of this scenario is to illustrate different work stealing patterns that arise in different intervals of time during the execution of a large-scale application. The interconnection topology follows the same policies as before: each site with a router, all the clusters of a site connected to the site router. The image **A** of Figure 6.16 shows the overall organization of the network topology, with dashed lines dividing the sites and each cluster representation with its respective name and number of processes allocated to it.

The total execution time of this application is 74 seconds. The image **A** of Figure 6.16 shows the work stealing requests that happened from the sixth to the sixteenth second of execution. In this time slice, most of the requests are performed between the *paraquad* and *paramount* clusters. The image **B** shows the time slice between the seconds 16 and 26, showing a higher number of steal requests inside the *rennes* site. The image **C** shows another time slice, from the seconds 26 and 36, with even more steal requests among the clusters and image **D** shows the time slice from the second 36 to 50. This last image has too many steal requests, causing problems in the perception of the network topology in the visualization base. This problem can be alleviated in the prototype by changing the transparency configuration of the links representation. Even so, the example shows an expected behavior from the KAAPI library, with more steal requests to the end of the application execution.

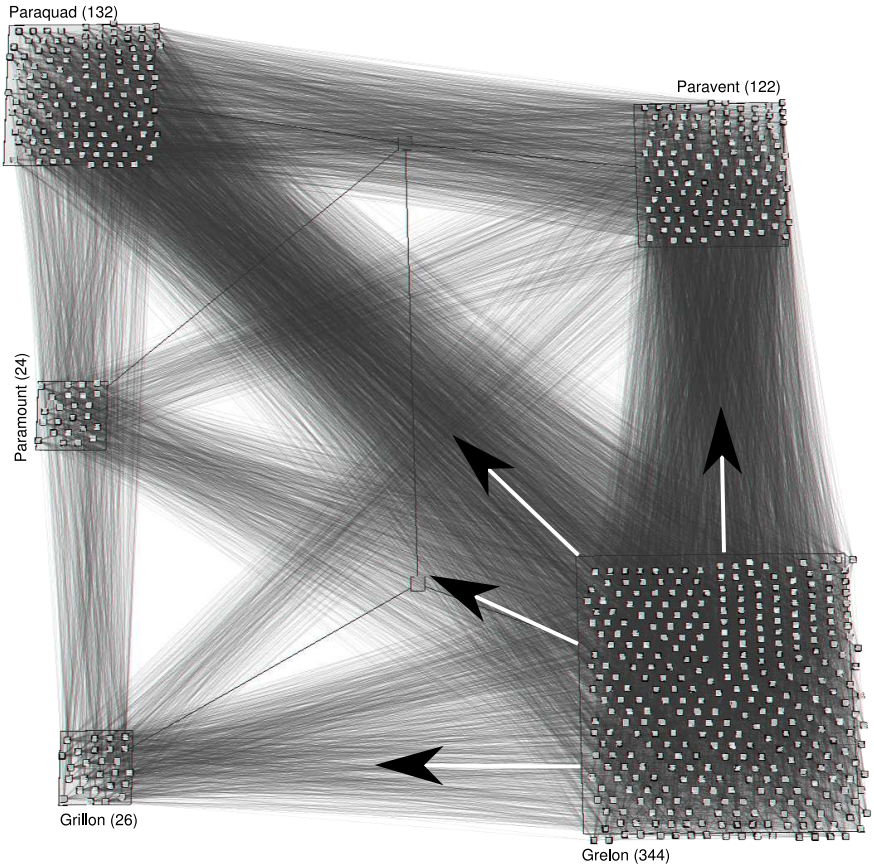


Figure 6.15 – Top-View generated by Triva showing the random work steal communication pattern of KAAPI.

6.3 Treemap Visualizations

The implementation of the Triva prototype included the development of the TimeSliceView and the Triva2DFrame (see Section 5.5 for details). As discussed, the 2D frame is developed to draw the treemap according to the execution of the Time-Slice algorithm and also the aggregation model implementation. A number of interaction mechanisms were also implemented in the prototype to facilitate the analysis. Examples are the use of the mouse wheel to navigate through the levels of the aggregated hierarchies; the use of two mouse buttons to select one or more states to analyze them separately; and the selection of the time slice on-the-fly.

An additional and important feature of the treemap rendering implementation is the use of the mouse pointer to highlight the hierarchy of a given leaf-node. The highlighting works by drawing a line in the border of the the leaf-node under the mouse pointer, complemented by rectangles in the parent nodes up to the root level. This drawings enable the identification of the hierarchy for a given leaf-node. Moreover, the prototype shows in the status bar of the window numerical information regarding the node under investigation and also the identifications of the

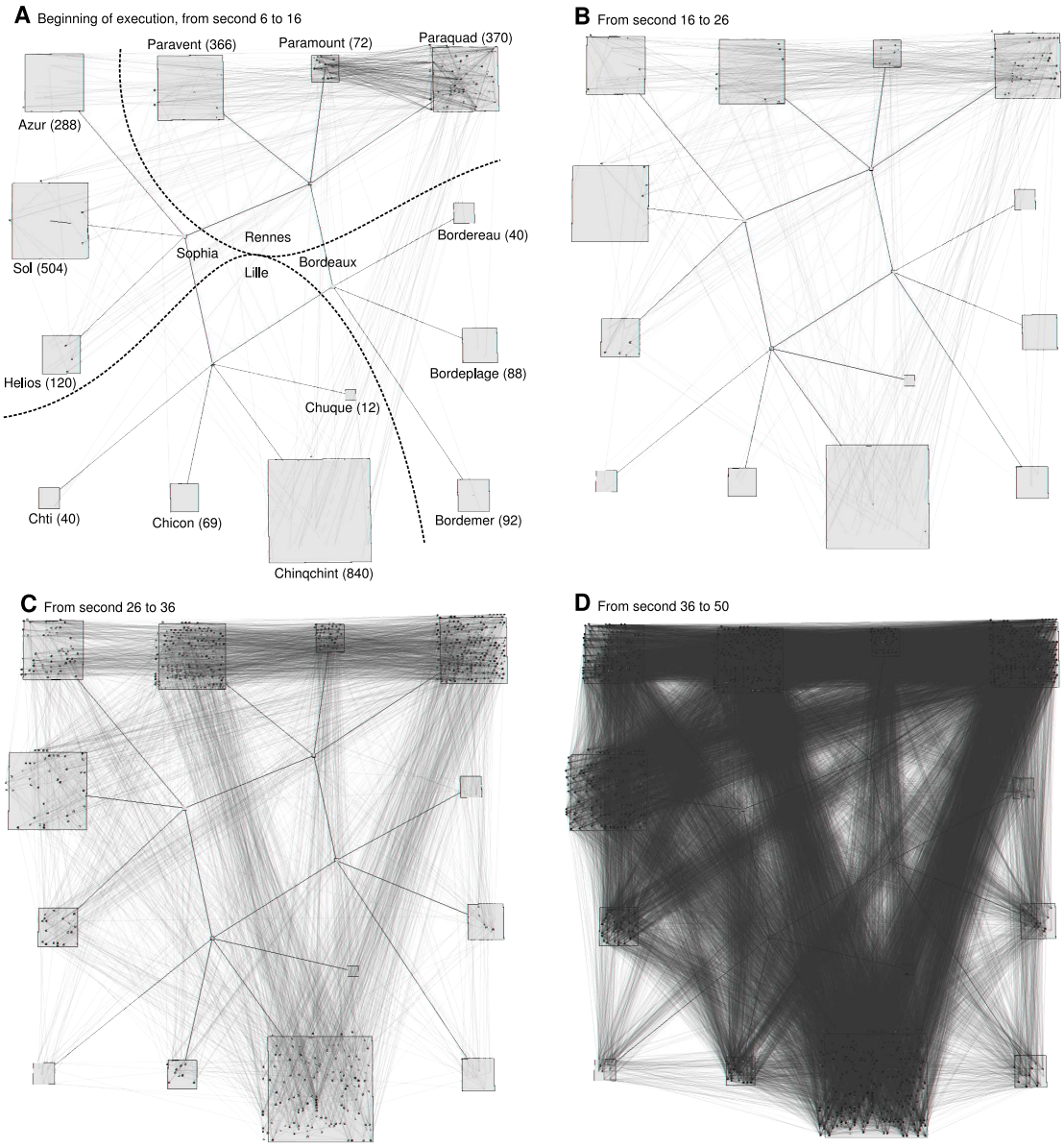


Figure 6.16 – Four top-views of an application executed in four Grid'5000 sites.

parents. Such interactive capabilities of the Triva prototype can be observed in the large treemap of Figure 6.17, with the dashed lines highlighting the hierarchical structure of a given leaf node.

This Section presents the results obtained with the treemap visualizations of synthetic and real trace data using the Triva Prototype. The treemaps presented in this Section were generated by the prototype and automatically exported to encapsulated postscript files. The main objective is to evaluate the potential of the proposed technique and to detect if the implementation is capable of reaching visualization scalability in large-scale situations.

We start with a general description of the treemap visualizations generated by the prototype, in the next sub-section. Afterwards, we present the visualization of a large-scale scenario created using synthetic trace files; and the analysis of different real-world scenarios using the KAAPI library and an example of visualization created with a MPI trace file.

6.3.1 Description of the Visualization

To describe the treemap visualizations created by the prototype, let us proceed to synthetic examples generated with the scripts described in the beginning of this Chapter. The first example is a hierarchy with three levels: Site – Cluster – Machine. There are two sites, each one with three clusters, each cluster with five machines. Therefore, the total number of machines is 30. Each machine can be in the Executing or Blocked state. Figure 6.17 depicts three treemaps that were generated with different properties. The two smaller treemaps on the left show only the Executing or the Blocked state, separately. Treemaps separated according to the state enable a direct comparison of which machines spent more time in a given state.

On the right side of Figure 6.17, the treemap shows in the same visualization the two states (Executing and Blocked) for all the machines. The inner dashed rectangle indicates the area reserved to one of the machines. The other dashed rectangles indicate the area that corresponds to the cluster that contains the machine and the site that cluster belongs to (the outermost dashed rectangle). These dashed rectangles were added manually to the treemap of the Figure since the method used in the prototype to highlight the hierarchy is not good for printing.

Moreover, we can notice that the visualization of more than one state (treemap on the right of Figure 6.17) enables a direct comparison among the machines but also the relationship among the states. This relationship is only correct if all the data being visualized is calculated based on the same metric. In this example, both Executing and Blocked states are calculated based on the amount of time in the time slice. Since both metrics are time-based, we can compare them. In terms of interactivity, the user of the prototype can go from the treemap on the right to the treemaps on the left just by clicking the state to be analyzed separately. The user can go back to the previous view with all the states with another click of the mouse.

The second example illustrates the treemap visualization in different levels considering aggregated values. The example is depicted on Figure 6.18 with four treemaps. The top-left treemap is the same of Figure 6.17, having the same hierarchy and the same numbers of machines, clusters and sites. This treemap is rendered in the **machine** level. As before, the Blocked and Executing states are always represented. The treemap on the top-right shows the aggregated values in the **cluster** level, the arrow between the top treemaps indicates that the area indicated on the left (the machine level) is summarized to the area on the right (the cluster level). In the middle of the top-left treemap there is a bold line that separates the two sites. The second arrow indicates the aggregation from the cluster level to the **site** level, shown on the bottom-left treemap. We can see on this treemap the two sites separation and the aggregated values of Executing and Blocked for each site. The last treemap on the bottom-right is generated using the maximum aggregation possible, where only the Executing and Blocked states are represented, considering all sites, clusters and machines below in the hierarchy.

The aggregated treemaps of Figure 6.18 enable the analysis of the states in different levels of the tree, showing their values for all the nodes. The top-right treemap of the Figure shows,

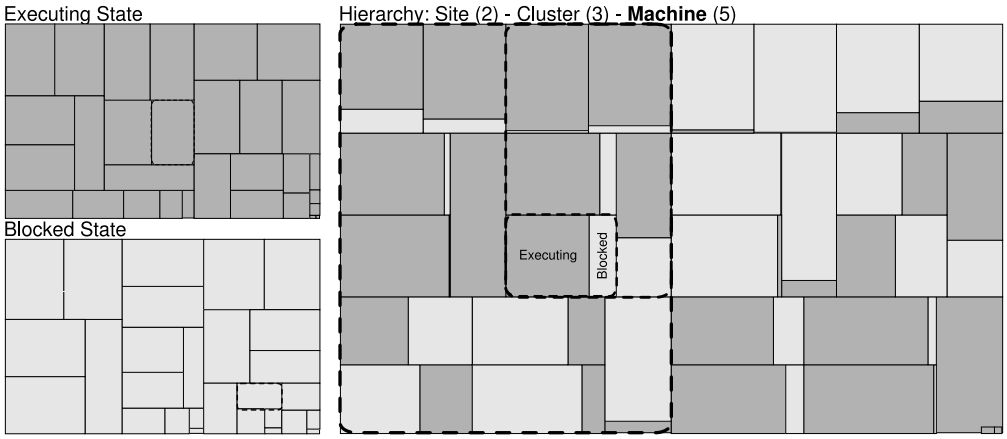


Figure 6.17 – Two squarified treemaps showing the states *Blocked* and *Executing* separately on the left, and on the same treemap on the right.

for instance, the Executing and Blocked state for the six clusters of the example (as indicated by the rounded dashed rectangles). We can clearly see the three clusters per site and the two sites. The values for the states for a cluster are calculated by the aggregation algorithm considering the Blocked and Executing states for the machines belonging to that cluster.

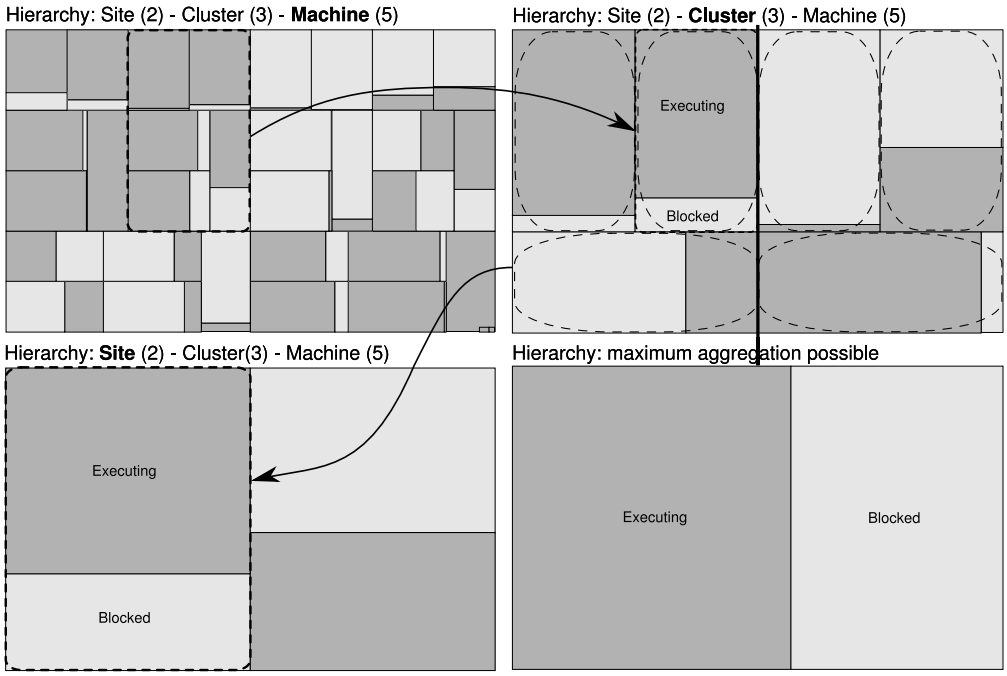


Figure 6.18 – Four treemaps to show the per-level aggregation of *Blocked* and *Executing* states.

Next sub-sections make extensive use of these representations, especially the aggregated

treemaps. For each of the scenarios, we explain the hierarchy used and the number of items per level. Most of the following examples use only one or two states for each of the leaves. The only exception for that is the MPI visualization, where the amount of time for three MPI operation is also represented.

6.3.2 Large-Scale Visualizations

One of the main benefits of the visual aggregation model, proposed in this thesis, is to easily analyze a large number of monitored entities on the same screen. In order to assess the visualization scalability of the approach, we generated a series of large-scale hierarchies using the synthetic trace generator. The objective is to show how the aggregation model behaves when dealing with so much information, and how the generated treemaps turn the data more understandable. A hierarchy composed by 100 thousand processors is analyzed in this Section. Figure 6.19 depicts the analysis of the chosen hierarchy, composed of four levels: Site, Cluster, Machine, and Processor. The hierarchy has 10 sites, each one with 10 clusters, each cluster with 10 machines, and each machine with 100 processors. Each processor can be in two possible states, represented in the Figure by the dark and light tonalities of gray.

The large-scale analysis using the prototype starts with the top-right treemap **A** of the Figure 6.19, in the **processor** level. In this treemap, there are 200 thousand rectangles: 100 thousand processors times the number of possible states, which is two. We can observe that some regions of this treemap are darker than others, allowing some analysis. However, any precise conclusion is hard to obtain with such treemap. The main reason is that treemap **A** has rectangles that are too small, turning out to be difficult to observe differences in sizes among two states of one single processor. The example is shown to present the limitation of the traditional treemap representation.

The white rectangle drawn manually in the treemap **A** of Figure 6.19 represents the space dedicated to one machine. Although it is hard to notice, there are 200 rectangles to represent the states of 100 processors inside this small area. Because of the fact that is hard to understand clearly the pattern of states to all 100 thousand processors, the user can interact with the prototype with the mouse wheel and show aggregated values for the **machine** level, as depicted in treemap **B** of the Figure. This treemap shows for each machine the two possible states. In this view, it is already possible to visually analyze major differences among the machines: some of them are significantly more often in one state than other, in the time slice considered to compute these treemaps. The highlighted area on the left of treemap **B**, shown through a zoom drawn manually, corresponds to the area highlighted through the white rectangle of treemap A.

Subsequent aggregations enable the user to visualize the traces in the **cluster** level, as depicted on treemap **C** of Figure 6.19, and in the **site** level, in treemap **D**. Treemap C shows the 100 clusters (10 per site). On the left part of this treemap, the black rectangle shows 10 clusters in the area dedicated for one site. The arrow beginning on this rectangle points to the aggregated values for that site, on treemap D. The maximum aggregation possible, shown on treemap **E**, enables a per state view of the available information, indicating that the state represented by the light gray color appears more than the other for the selected time slice.

Observing the example of Figure 6.19, we can see the benefits brought by the aggregation algorithm. Its implementation in the Triva prototype enables the visualization of several thou-

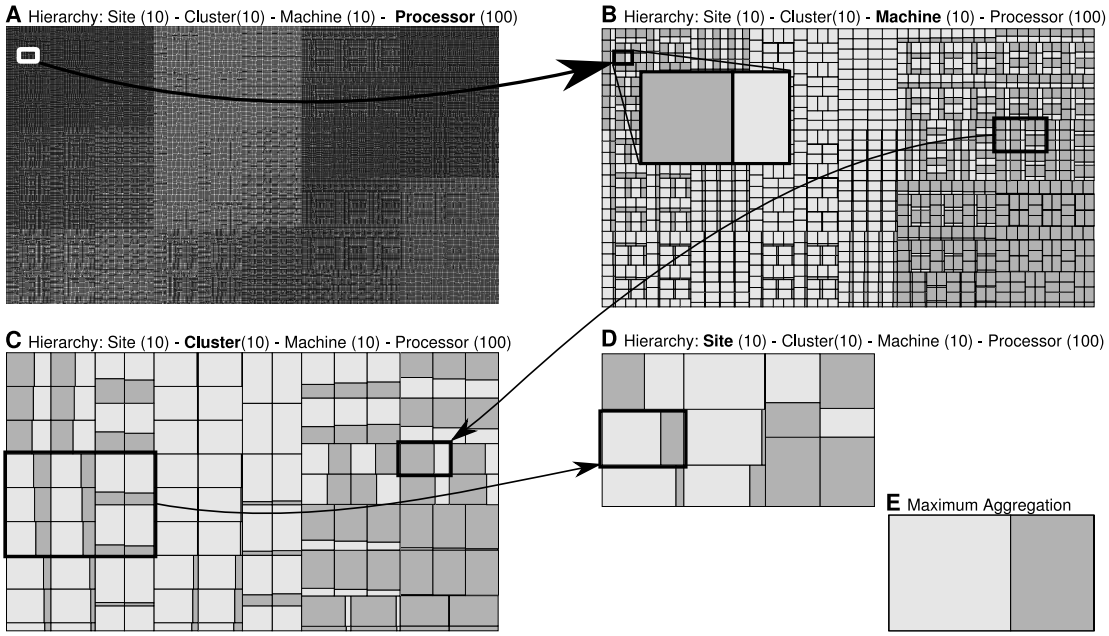


Figure 6.19 – Normal (A) and four aggregated treemap visualizations (B – E) of two states for 100 thousand processors (based on synthetic trace).

sands monitored entities, possibly with the presence of a number of states. The example also illustrates that the performed aggregations enable a better understanding of the behavior of entities in different levels, by interactively grouping the states in the hierarchy. Moreover, we can also observe that even among the aggregated treemaps, the same general behavior can be visualized, with a much simpler representation.

6.3.3 KAAPI Work Stealing Analysis

This Section presents the treemap visualizations of the Triva prototype using as input trace files generated by the KAAPI library. As stated, these traces register the behavior of the work stealing activities of the library to provide load balancing to the parallel applications. The traces were obtained during the execution of KAAPI applications in the Grid'5000 platform. We selected four different scenarios to explain the possible analysis that can be performed with the prototype Triva. Each scenario has a different configuration of resource allocation to execute the applications, and a different number of KAAPI processes involved. For all the treemaps of this Section, the light gray color of rectangles indicates the RUN state, and the darker gray indicates the RSTEAL state, for every K-Process of a KAAPI application, or for every level when an aggregated treemap is presented.

Scenario A: 200 processes, 200 machines, two sites

The first scenario is the execution of a KAAPI application composed of 200 processes. Each process is allocated to one machine of the Grid'5000 platform, resulting in an allocation of 200 machines divided equally in two sites of the grid: *rennes* and *nancy*. The former site allocation is the following: 61 machines from cluster *paravent*, 33 from *paraquad*, and 6 from *paramount*; the later is: 86 from *grelon*, and 14 from *grillon*. The treemaps depicted on Figure 6.20 illustrate the behavior that the application showed during the execution on that allocation, in three different time slices.

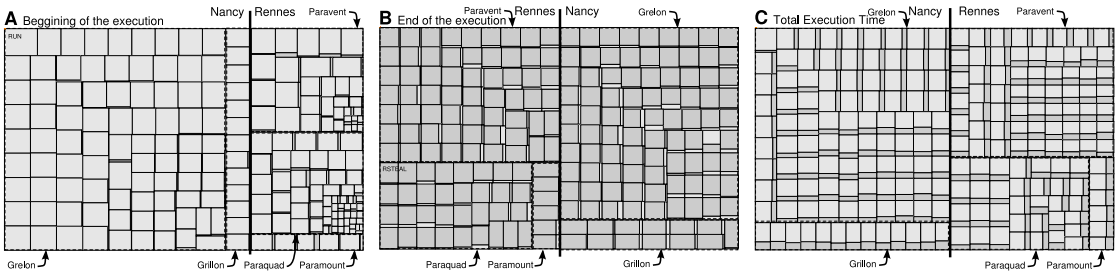


Figure 6.20 – KAAPI Scenario A with an application composed of 200 processes.

The treemap **A** of the Figure 6.20 is computed using a time slice that corresponds to the beginning of the application. During this period, we can observe that most of the K-Process are actually running and not spending time trying to steal tasks. Since the application was launched in the *nancy* site, we can observe that the K-Processes belonging to this site occupy more space when compared to the space occupied by the *rennes* site. Treemap **B** is computed based on a time slice of the end of the execution. We can observe that in the end of execution, the K-Processes spent more time trying to steal tasks from others processes. This is a normal situation, since when the program approaches the end, new tasks to execute become rare. The treemap **C** is computed considering all the execution time for the application. By doing this broad analysis with a large time slice, we can observe global patterns that might arise for a set of K-Processes. This actually happens in this example, since this treemap shows that most of K-Processes maintain the same relation between time spent in Run and RSteal states. This is observed through the sizes of each state for the processes.

Another thing that is possible to conclude analyzing treemap C of Figure 6.20 is the load balancing between the two sites. Since this treemap is computed using the total execution time and each site has an equal number of allocated machines, we can argue that an ideal situation for this scenario will be that the area occupied by each of the site in the visualization should be the same. This will indicate that an ideal load balancing is achieved by the KAAPI work stealing algorithm. The treemap C indicates that the area for the *nancy* site is slightly bigger than the area for the *rennes* site, letting us conclude that an ideal load balancing is not achieved. The explanation for such behavior can be that the application is launched in one machine of the *nancy* site, allowing the K-Processes of this site to start the execution of tasks before the processes of the *rennes* site. Even so, considering that the areas for each site in the treemap are only slightly different, we can argue that the load balancing achieved by the work stealing is of

good quality.

Scenario B: 400 processes, 50 machines, one site

The second scenario with KAAPI traces is an application composed of 400 processes executed in 50 machines of the *bordeaux* site of Grid'5000. In the experiment, the allocation is composed of 23 machines from the *bordemer* cluster, 22 from the *bordeplage* and 5 from the *bordereau*. The two treemaps of Figure 6.21 are computed using the traces from this scenario. The bold lines in both treemaps separate the three clusters involved in the execution.

The treemap **A** of Figure 6.21 shows all the processes with the Run and RSteal states. We can notice in this treemap that there are some K-Processes that spent an unusual amount of time in the RSteal state when compared to the others processes. This might indicate a problem in the machines that execute those processes, since each machine received eight K-Processes to execute. The treemap **B**, on the right, is computed using as parameter the same time slice but only the RSteal state. Treemap B also shows the amount of seconds for the larger areas, indicating that processes with unusual behavior spent around 40 seconds trying to steal tasks from others. Since only one Grid'5000 site was used and the allocated clusters are interconnected with local networks, the probable cause of these anomalies should not be attributed to the network. The only remaining explanation for such behavior is related to the amount of K-Processes executed per each machine.

The *bordemer* and *bordeplage* clusters have machines with 2 CPUs. The *bordereau* cluster has machines with 4 CPUs. As stated, there is 400 processes and 50 machines on this scenario, resulting in 8 processes per machine. We can observe in the Figure 6.21 that only K-Processes in clusters with 2 CPUs ended with an unusual behavior. A possible explanation is the overload of processes on those machines when compared to the machines of the *bordereau* cluster, with 4 CPUs each, that did not show the odd behavior.

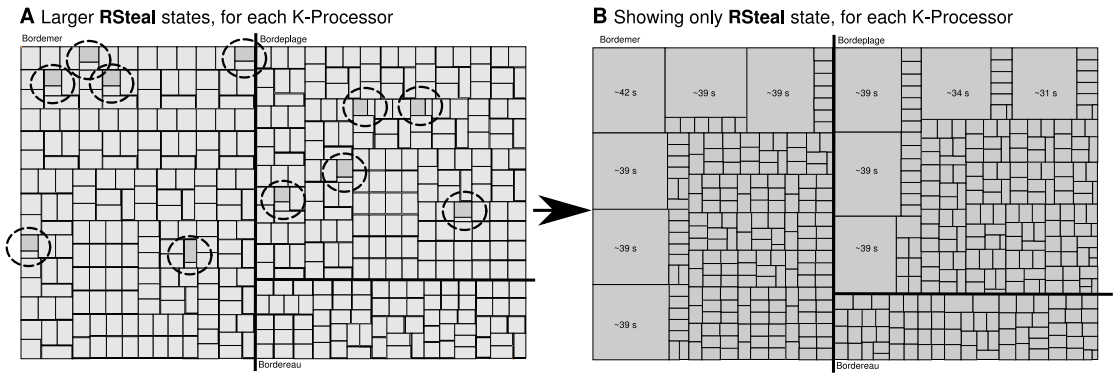


Figure 6.21 – KAAPI Scenario B with an application composed of 400 processes.

Scenario C: 2900 processes, 310 processors, four sites

The third scenario is a KAAPI application with 2900 processes, executed in 310 machines that were allocated in clusters of four Grid'5000 sites. The machine allocation is as follows: 60 machines from *lille* site (41 - *chinqchint*, 10 - *chti*, 3 - *chuque*, 6 - *chicon*); 100 from *rennes* (61 - *paravent*, 6 - *paramount*, 33 - *paraquad*); 50 from *bordeaux* (5 - *bordereau*, 22 - *bordeplage*, 23 - *bordermer*); and 100 from *sophia* site (48 - *azur*, 42 - *sol*, 10 - *helios*). The objective of this scenario is to show that the prototype is able to deal with large trace files with events from an application executed in a real platform. As stated, there are two possible states for each of the 2900 processes. This results in a treemap that must draw 5800 rectangles. Figure 6.22 illustrates, in treemap **A**, all these rectangles that together represent the behavior of 2900 K-Processes. Bold lines indicate Grid'5000 cluster division.

The treemap **A** of Figure 6.22 shows the Run and RSteal states for all the processes. The time slice used to compute the treemap is the total execution time of the application. We can notice in this scenario that the amount of time spent with work stealing requests is very small. In the treemap **A**, it is difficult to perceive the rectangles that represent the state RSteal. The treemap **B**, on the top-right, depicts only the Run state for all the processes. Analyzing the screenshot, it is possible to conclude that almost all K-Processes spent the same amount of time executing tasks. The only exception is the K-Processes located in the *chti* cluster, in the bottom-middle region of treemap **B**. They have smaller rectangles indicating less time in the Run state.

The treemap **C** of Figure 6.22 shows, on the other hand, only the RSteal state for all K-Processes. Differently from the Run state, here we can notice different rectangle sizes indicating that some processes spent more time stealing tasks than others. This might indicate for example which processes are executed on faster machines, finishing their tasks more frequently; or can indicate processes that execute more unsuccessful steal requests when idle. The treemap *D* of the same Figure shows the RSteal state, but now aggregated by machine. Analyzing this treemap, we are able to detect instantaneously which machines spent more stealing. The white rectangles on treemap C and D indicate an example of aggregation of the RSteal states of ten K-Processes to the machine where they executed. A possible reason for such behavior is the work propagation at the beggining of the execution.

Scenario D: 188 processes, 188 machines, five sites

The fourth scenario is an application of 188 processes, executed in 188 machines, distributed in five sites of Grid'5000 including the cluster from Porto Alegre, Brazil. There are 13 machines allocated from the cluster *xiru*, at *portoalegre* site; 2 from *bordereau*, 17 from *bordemer*, and 6 from *bordeplage*, at *bordeaux* site; 45 from *pastel*, 5 from *violette*, at *toulouse*; 14 from *paramount*, 36 from *paraquad*, at *rennes*; and finally 50 from *grelon* cluster at *nancy* site. The Figure 6.23 shows two treemaps calculated with the traces generated by the application of this scenario.

The treemap **A** shows the Run and RSteal states for all the 188 processes. Almost all processes show the same behavior, with a bigger Run state (the light gray areas) compared to the RSteal state (the dark gray areas). The only exception appears in the K-processes executed in the *portoalegre* site, highlighted manually with the dashed circle. Observing this treemap, we

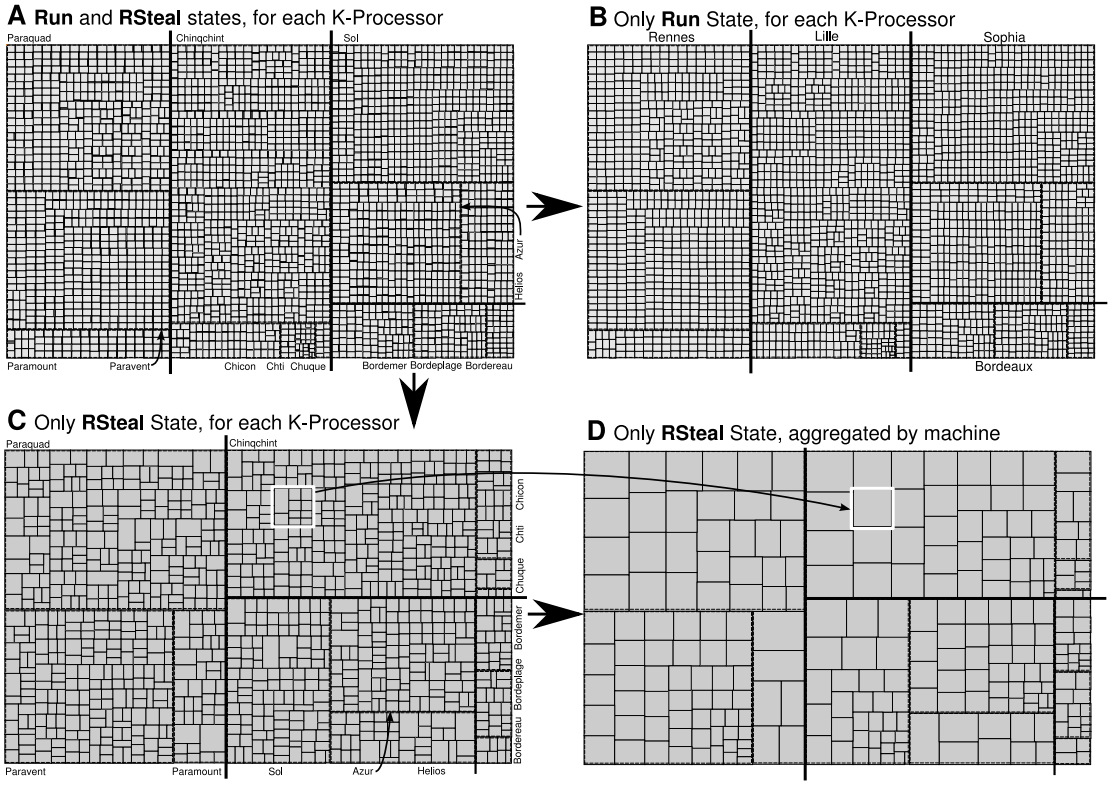


Figure 6.22 – KAAPI Scenario C with an application of 2900 processes.

notice that these processes spent more time stealing tasks than the processes from other sites. The treemap **B**, on the right, shows the same time slice and the same processes, but only the RSteal state. Here, the difference in the time spent stealing tasks become even more evident. We believe that the main reason behind this behavior comes from the interconnection of the sites. The *portoalegre* site is located in Brazil, and its connection with the Grid'5000 is made through a Virtual Private Network (VPN) that is maintained through the internet. The latency of this interconnection compared to the general latency among Grid'5000 sites located in France is significant. The traditional work stealing algorithm inside KAAPI do not differentiate from which processes a given process will try to steal. This, in a heterogeneous interconnection environment, may lead to more time spent trying to steal, as indicated by the treemap computed with our Time-Slice technique.

Generally speaking, the Time-Slice algorithm combined with the aggregation model of this thesis enables an easy identification of performance issues when comparing the behavior of processes of a parallel application. The aggregation model brings these advantages to large-scale situations, no matter how many processes are involved in the analysis. The only step necessary to make both proposals work well in large-scale environments is to set a proper hierarchy with at least some levels. The hierarchies used through out the KAAPI scenarios have five levels, which already allows the analysis of several thousands of processes.

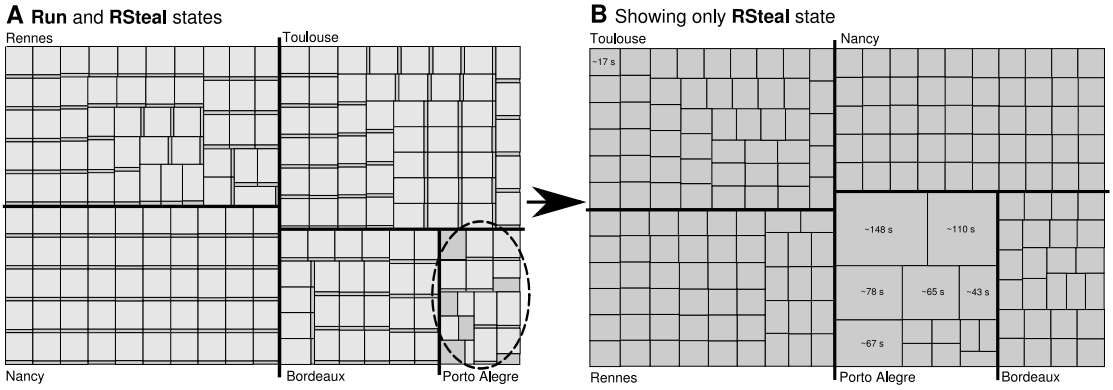


Figure 6.23 – KAAPI Scenario D with an application composed of 188 processes.

6.3.4 MPI Operations Analysis

The experiment described here uses traces generated during the execution of a MPI application. The traces were described in the beginning of the Chapter, recording the execution of MPI operations. The objective here is to show that the Triva prototype is also capable of analyzing MPI applications, because of the use of the generic Pajé file format as input.

The scenario for the MPI experiment is as follows. We executed the EP application, part of the NAS benchmark, using 32 processes in a cluster where each node has 8 processors. The tracing mechanisms registered the following MPI operations during the execution of the MPI application: *MPI_Init*, *MPI_Barrier* and *MPI_AllReduce*. For each of these operations, there is one state of the same name. We use the state Running to indicate the time spent outside of MPI operations. The hierarchy defined in the traces is flat, only with the MPI process level.

The analysis of the scenario is depicted on the treemaps of Figure 6.24. The treemap **A** shows the amount of time spent in each of the states. We can notice that there are some differences among the processes, as illustrated by the dashed rectangles of two MPI processes. The square on the right of the treemap A shows a zoom to the MPI Process rank 21, where the correspondence of gray tonalities and the states are noted. On the Figure, the treemap **B** shows the amount of time all the processes spent in the *MPI_Init* state. The numbers in the rectangles indicate the amount of time for the process, an information that can be obtained by pointing the mouse to that region of the window in the Triva prototype. We can notice significant differences of time spent on the init state. Treemap **C** has the same single state rendering, but here using the time of the *MPI_Barrier* operation. We can observe that the behavior of the init and barrier state are very similar, possibly indicating that the implementation of the MPI init operation is close to the implementation of a barrier. On the bottom of the Figure, the treemap shows the maximum aggregation considering all the 32 processes. Analyzing this aggregated view, it is possible to observe that the time spent in MPI operations is greater than the Running state, where the code of the application is probably placed.

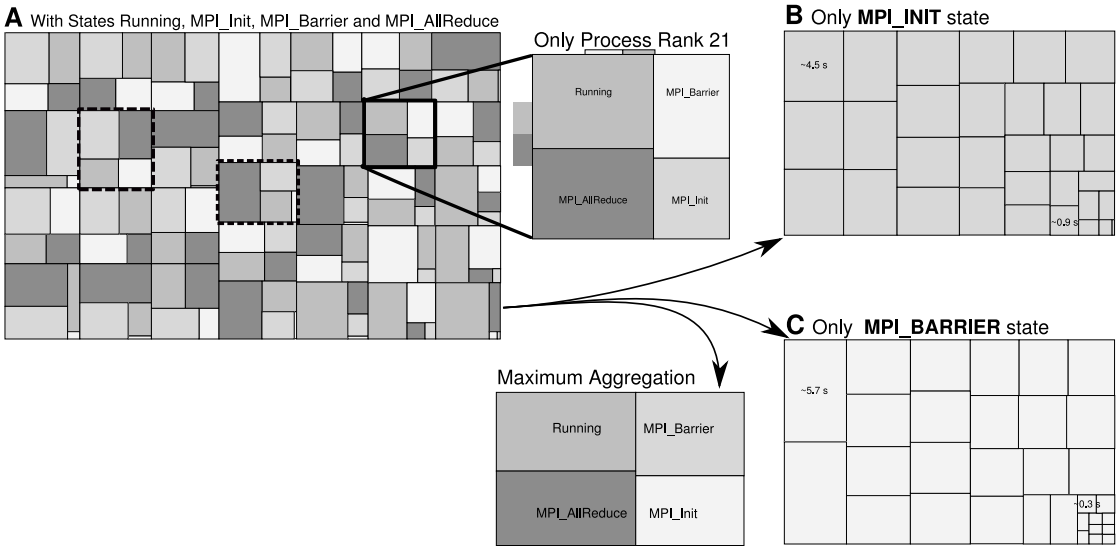


Figure 6.24 – Behavior of the EP application of the NAS Benchmark using treemaps and the aggregation technique.

Chapter 7

Conclusion and Future Work

Traditional visualization schemes for the analysis of parallel applications are designed to handle monitoring data collected at small scale and in regular environments. The necessity of visualization techniques for the analysis of parallel applications on highly distributed systems, such as grids, motivated this work. Two particular problems of the traditional analysis of applications have been identified in this thesis.

The first one is the impact of the network interconnection on the execution of parallel applications. This impact must be outlined in the analysis in order to better understand and improve the application performance. Traditional visualization techniques, such as the space-time representation, are widely used for the analysis of applications. These techniques, however, cannot show in the same screen the network topology and the monitoring data from the application. This might lead to wrong conclusions during the detection of performance issues of applications. The second problem is the visualization scalability of traditional techniques. Usually, the number of monitored entities that can be analyzed on the same screen is often limited to the vertical size of computer screens. Space-time representations are a clear example of this matter, being not well suited to grid applications composed of thousands of processes.

The main idea behind this thesis is to explore information visualization techniques that can be used to visualize parallel applications. Our first approach is the three dimensional visualization, where the base of this visualization is used to detail the resource/application organization, and the third axis to show the evolution of the application through time. We have implemented three different base configurations within the 3D approach: the representation of the network interconnection with application behavior; the representation of the application communication pattern and another to observe processes balance on the resources.

The second approach is the visual aggregation model, where the scalability problems of traditional visualization tools are solved through a combination of the treemap technique and the Time-Slice algorithm. This algorithm takes into account intervals of time to generate values and inject them in a hierarchical organization of the application being analyzed. This structure is then passed out to the treemap technique that renders the visualization. The visualization scalability is achieved through the aggregation model, where the levels of the hierarchy are explored to create intermediary information that can be used to help the analysis from the most detailed view to the most general one.

Both approaches are implemented in a prototype called Triva, developed using a 3D rendering engine called Ogre, GraphViz, some of the Pajé components, and an implementation of the squarified treemaps from scratch. The prototype has a reading mechanism that links it with the DIMVisual integration library, capable of integrating monitoring data from different sensors and formats. Synthetic traces, but also real trace data from KAAPI and MPI applications are used to validate the approaches and the implementation. KAAPI traces used in this thesis were collected in the Grid'5000 platform. Although the prototype validation is attached to these traces, the use of the generic Pajé file format allows the extension of the benefits brought by the implementation to other fields and applications, from resource visualization to other types of communication libraries.

The obtained results are promising. The three-dimensional visualization, analyzed in the results Chapter, allows a better understanding of applications communications in contrast with the network topology. We were able to show in different time slices that the work stealing could benefit from more locality, since the current implementation of KAAPI do not take into account network information to perform work stealing requests. On the other hand, the results obtained with the visual aggregation model implementation allowed the visualization of the states of 100 thousand processors, generated synthetically. The treemaps defined by the Time-Slice algorithm were also generated using real trace data from KAAPI and MPI applications. We were able to identify in KAAPI traces different aspects, such as a different behavior in stealing mechanisms presented by some processes, load-balancing efficiency considering all the execution time, and the analysis of a large-scale KAAPI application, composed of almost 3 thousand processes in Grid'5000.

In summary, the main achievements of this thesis are the proposal of the 3D approach, the visual aggregation model combined with the Time-Slice technique and the Triva prototype implementation. Other achievements include the interaction between KAAPI and the prototype, allowing the analysis of KAAPI work stealing activities.

Next Section presents the publications that came from this thesis. Section 7.2 discusses the perspectives and implications of this thesis.

7.1 Publications

Some results of the thesis were published in the following papers:

- **Visual Mapping of Program Components to Resources Representation: a 3D Analysis of Grid Parallel Applications**. The 21st Symposium on Computer Architecture and High Performance Computing, SBAC-PAD. 2009. IEEE Press. Sao Paulo, Brazil.

  – This paper presents the use of the three-dimensional approach to map parallel applications components on top of a resource representation. The paper describes the abstract model that generate this 3D configuration, showing at the end some examples of KAAPI parallel applications visualized together with the Grid'5000 network topology.

- **Visualization of Parallel Applications: Results of an International Collaboration**. Colloque d'Informatique: Brésil / INRIA, Coopérations, Avancées et Défis. Colibri 2009. Bento Goncalves, Brazil.
  - This 4-page paper presents the overall proposal of this thesis, including the two visualization models. The paper is also focused on presenting the international collaboration between UFRGS and INPG, through the co-advising agreement of the student.
- **Towards Visualization Scalability through Time Intervals and Hierarchical Organization of Monitoring Data**. The 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID, 2009. Shanghai, China.
  - This paper presents the Time-Slice algorithm when used to summarize states of a parallel application. The paper also presents basic concepts of the treemap representation and how they are used to provide a visualization for the hierarchies generated with the Time-Slice algorithm. The hypothesis of the paper is validated with the visualization of KAAPI traces composed of almost three thousands processes.
- **3D Approach to the Visualization of Parallel Applications and Grid Monitoring Information**. The 9th IEEE/ACM International Conference on Grid Computing, GRID, 2008. Tsukuba, Japan.
  - The paper presents the overall view and general structure of the 3D approach. Besides presenting the generic abstract model to create such representations, the paper also detail the visualization of synthetic and well-known communication patterns, but also the visualization of KAAPI traces.
- **Triva: Interactive 3D Visualization for Performance Analysis of Parallel Applications**. Accepted in the Future Generation Computer Systems Journal, of Elsevier.
  - This 24-page journal paper presents the 3D approach, the abstract component model and results. It is strongly based on Chapter 3 of this thesis, with some three-dimensional visualizations obtained with the prototype as presented in Chapter 6.

7.2 Implications and Perspectives

There are several perspectives considering the two visualization models proposed in this thesis. The three-dimensional model, today, shows every detail about all the monitoring entities. A possible evolution of this behavior is the view of aggregated data. Therefore, instead of showing all the links in a time interval, the program would show just one link that represents the aggregated information. Visually, the link could be rendered according to the information it represents: bigger when more information is contained within, smaller otherwise. Such representation could also be extended to the states of a monitored entity.

Many other types of information for the Time-Slice algorithm still need to be studied. We basically analyzed only states in our results, because most of our traces are composed by states

for the processes. Other information, such as links, variables, events, must also be studied and explored. Particular investigation should be conducted in the case of the links, where we left open in our model to which node attribute a summary value: the origin or the destination. Depending on the type of information being evaluated, a situation may be better than another.

The evolution of the aggregation model with other aggregation functions is also possible. Although we discussed briefly the use of other functions, we used in our results only the addition aggregation. Functions such as max, min, mean must be studied, particularly when other type of summary data is generated by the Time-Slice technique. User defined aggregation functions, based on the available monitoring data must also be analyzed.

For the 3D approach, a possible perspective is to improve the visual mapping between the network topology and the communication pattern of the application, through the use of curved lines to represent communications. Besides the matching that is already modeled in the 3D approach, the abstract treatment of data should also consider other types of information, such as the size of links and nodes. This should be reflected directly in the visual representation. Generally speaking, this perspective means that a graph must serve as a base to the rendering of another graph. The representation could also be guided according to routing rules of the interconnection, when they are available from the execution environment. The 3D approach might also be used to the visualization of parallel applications in many-core chips, where a network-on-chip is present inside the processor.

A third possible evolution is the merging of the hierarchical organization of monitoring data with a graph representation. This could be explored in the visual aggregation model by defining in every level of the hierarchy, a graph to represent interactions. The links of this graph could be annotated with aggregated data, as we already do in the Time-Slice technique. An example of application for such evolution is the merging of processes of a parallel application.

Perhaps the most significant implication of this work is the study of information visualization techniques applied to the parallel application analysis. Since we used a study like this as inspiration for the thesis, we think that it can be continuously faced as motivation for new work.

Appendix A

Extended Abstract in Portuguese

The Portuguese title for this thesis is *"Alguns Modelos de Visualização aplicados para a Análise de Aplicações Paralelas"*. The extended Portuguese abstract is presented here to fulfill the requirements established in the *co-tutelle* agreement of the author.

The abstract of this appendix is basically a Portuguese translation of the more important parts of the English version of this thesis, especially the introduction of the chapters and main concepts of the proposed visualization models. Two experimental scenarios of the main document were selected for the sake of demonstrating some results in this extended abstract.

A.1 Introdução

Sistemas distribuídos consistem basicamente em hardware e software que contêm mais de uma única unidade de processamento [19]. Nestes sistemas, os processadores são interconectados e comunicam através de uma rede. Os programas de computador são quebrados em várias partes e devem lidar com diferentes níveis de paralelismo e com algoritmos de comunicação, como passagens de mensagem e memória compartilhada. Um exemplo de sistema distribuído é chamado de Grid [30]. Estes tipo de sistema é estruturado em organizações virtuais [29], possivelmente compostas por milhares de máquinas distribuídas geograficamente. Dois exemplos de Grid são o Grid'5000 francês [12] e o TeraGrid americano [16].

Características compartilhadas por quase todas as plataformas Grid são dinamismo, heterogeneidade de recursos e software, e presença de múltiplos domínios administrativos. Dinamismo significa que os recursos que participam de um Grid pode se tornar indisponíveis a qualquer hora, sem nenhuma notificação de que isso pode acontecer em um determinado momento. Aplicações paralelas devem lidar com isso no nível da aplicação ou através de uma biblioteca capaz de lidar com flutuações na quantidade de recursos disponíveis. A heterogeneidade significa que diferentes configurações de recursos pode estar presentes na mesma plataforma Grid. Isto também é válido para bibliotecas de software. Um Grid pode estar espalhado por diferentes domínios administrativos, cada parte mantida independentemente por seus administradores. Além destas características, um Grid também pode ter uma rede de interconexão complexa e ser facilmente escalável quanto aos seus recursos.

A interconexão entre os recursos de um Grid pode ser composta por diferentes tipos de rede. Ela pode ser composta por tecnologias Ethernet, Myrinet, Infiniband, ou fibra óptica. Um exemplo de Grid com vários tipos de interconexão são os chamados Desktop Grids [48], como os projetos BOINC [1] e Seti@Home [2], onde a interconexão é em geral feita através da internet. Outro exemplo que evidencia a presença de múltiplos tipos de interconexão é um Grid composto por clusters, onde uma hierarquia de interconexão forte é usada para conectar clusters homogêneos [12]. A presença de vários tipos de interconexão é um reflexo natural da heterogeneidade e da distribuição geográfica de Grids. Estes aspectos impõe uma rede mais complexa, um número maior de saltos para comunicação entre processos e latência e largura de banda variáveis e diferentes ao longo do tempo.

Plataformas Grid são também facilmente escaláveis, de uma forma que novos recursos podem ser indefinidamente adicionados apenas conectando eles aos participantes existentes. Normalmente, estas adições de recursos trazem mais heterogeneidade e aumentam a complexidade da rede. Atualmente, existem Grids globais que são compostos por milhares de computadores, como o exemplo do projeto BOINC. Outro exemplo de quão fácil é a adição de novos recursos a um Grid é o caso do Grid'5000, onde novos clusters são adicionados ao backbone principal da plataforma. A escalabilidade destas plataformas é uma boa característica do ponto de vista das aplicações paralelas, que necessitam cada vez mais de mais recursos computacionais.

Todas essas características de plataformas Grid influenciam diretamente o comportamento das aplicações paralelas durante o seu desenvolvimento e execução. Por causa disto, é importante para o desenvolvedor entender o impacto do sistema distribuído sobre a aplicação. Um exemplo disso é a análise de aplicações considerando a topologia da rede. A aplicação pode ter

um melhor ou pior desempenho dependendo de quais recursos foram escolhidos e a interconexão entre eles. Esta influência é ainda mais evidente quando os aspectos da rede são considerados, como a latência e a largura de banda, em aplicações que são limitadas pela rede. A escalabilidade de um Grid é outro aspecto que também influencia diretamente o comportamento das aplicações paralelas, uma vez que a disponibilidade de novos recursos para a aplicação não indica sempre que um melhor desempenho será alcançado.

Considerando estas situações, podemos perceber que é importante analisar o comportamento das aplicações paralelas com informações do Grid. Esta análise pode ajudar desenvolvedores a entender o impacto da topologia da rede na aplicação por exemplo. Contrastando o padrão de comunicação da aplicação com a topologia da rede pode dar dicas ao desenvolvedor de como adaptar a aplicação para melhor explorar tal interconexão. Além disso, se a rede é hierarquicamente organizada, as aplicações podem seguir a hierarquia da rede para evitar gargalos e outros problemas de desempenho se a aplicação não é estruturada hierarquicamente. Uma boa análise também deve levar a conclusões sobre todos os processos da aplicação, incluindo padrões locais e globais que podem aparecer entre eles. Se existe uma grande quantidade de processos, a análise deve ser capaz de gerar resultados sobre todos eles.

A visualização é uma forma de realizar a análise de aplicações paralelas. Ela tem sido bastante utilizada nos últimos 30 anos para entender e observar aplicações que são focadas em diferentes níveis de paralelismo. A forma mais tradicional de visualização acontece através de uma adaptação de gráficos Gantt [79], também conhecido como gráficos espaço-tempo. Estas visualizações listam os componentes da aplicação verticalmente e sua evolução no tempo é demonstrado no eixo horizontal. Exemplos de ferramentas que oferecem este tipo de análise são o Pajé [22], Vampir [60] entre outras [46, 63, 5]. Gráficos espaço-tempo são bastante usados em plataformas existentes, como clusters, onde os dados são simples e uniformes.

Muitas dessas ferramentas de visualização foram adaptadas para observar o comportamento de aplicações em sistemas distribuídos como Grid. Geralmente elas continuam usando as mesmas técnicas de visualização. Considerando somente representações espaço-tempo, o primeiro problema que surge é que elas não podem representar, juntamente com os dados da aplicação, a complexa topologia de rede de plataformas Grid. Como discutido, o impacto dessa topologia não pode ser excluído de uma análise de aplicação quando uma interconexão complicada existe entre os recursos. O segundo problema é relacionado com a escalabilidade de visualização de gráficos espaço-tempo. Usando tais representações, o número de componentes da aplicação que podem ser visualizados uma tela de computador é limitado à resolução vertical da tela.

Esta tese tenta resolver estes problemas encontrados em técnicas de visualização tradicionais para aplicações paralelas. A idéia principal dos esforços consiste em explorar técnicas da área de visualização da informação e tentar aplicá-las no contexto de análise de programas paralelos. Levando em conta isto, esta tese propõe dois modelos de visualização: o de três dimensões e o modelo de agregação visual. O primeiro pode ser utilizado para analisar aplicações levando-se em conta a topologia da rede dos recursos. A visualização em si é composta por três dimensões, onde duas são usadas para mostrar a topologia e a terceira é usada para representar o tempo. O segundo modelo pode ser usado para analisar aplicações paralelas com uma grande quantidade de processos. Ela explora uma organização hierárquica dos dados de monitoramento e uma técnica de visualização chamada Treemap para representar visualmente a hierarquia. Os dois

modelos representam uma nova forma de analisar aplicação paralelas visualmente, uma vez que eles foram concebidos para larga-escala e sistemas distribuídos complexos, como Grids.

Alguns dos conceitos desta tese foram publicados e um artigo está em processo de avaliação.

Este resumo estendido está organizado em cinco seções, descritos a seguir:

Seção A.2: O Modelo Tri-Dimensional

Esta seção apresenta o primeiro modelo desta tese, composto da abordagem em três dimensões. Nele, descrevemos a concepção do modelo visual e uma visão geral dos componentes abstratos capaz de gerar visualizações 3D.

Seção A.3: Modelo Visual de Agregação

A seção apresenta a concepção do algoritmo de fatia de tempo para a transformação do comportamento de uma aplicação em uma hierarquia, além do modelo de agregação usado para se atingir escalabilidade visual no uso de representações Treemap.

Seção A.4: O Protótipo Triva

Esta seção apresenta o protótipo desenvolvido ao longo desta tese. O foco da descrição neste resumo fica na parte da visão geral dos componentes que fazem parte da implementação.

Seção A.5: Resultados e Avaliação

A seção apresenta os resultados obtidos com o protótipo Triva na avaliação dos modelos propostos. Dois cenários são apresentados: um relacionado ao tri-dimensional, e outro ao modelo de agregação visual.

Seção A.6: Conclusão

Os principais resultados são relembrados e as perspectivas delineadas.

A.2 O Modelo Tri-dimensional

O desempenho de aplicações Grid está relacionado às características da rede de interconexão [49]. Quando os recursos tem uma forte hierarquia entre eles, a escolha dos recursos dados a uma aplicação pode ser decisivo para o desempenho e também para o entendimento da aplicação. Sem informações da topologia da rede, o analista pode não ser capaz de perceber que eventuais problemas na aplicação são devido a limitações do nível da rede. As decisões tomadas por uma visualização tradicional da aplicação, neste caso, podem levar a conclusões erradas sobre o mau desempenho. Sendo assim, se fossemos capazes de analisar a aplicação levando-se em conta características da rede, nós veríamos mais claramente as razões do comportamento da aplicação.

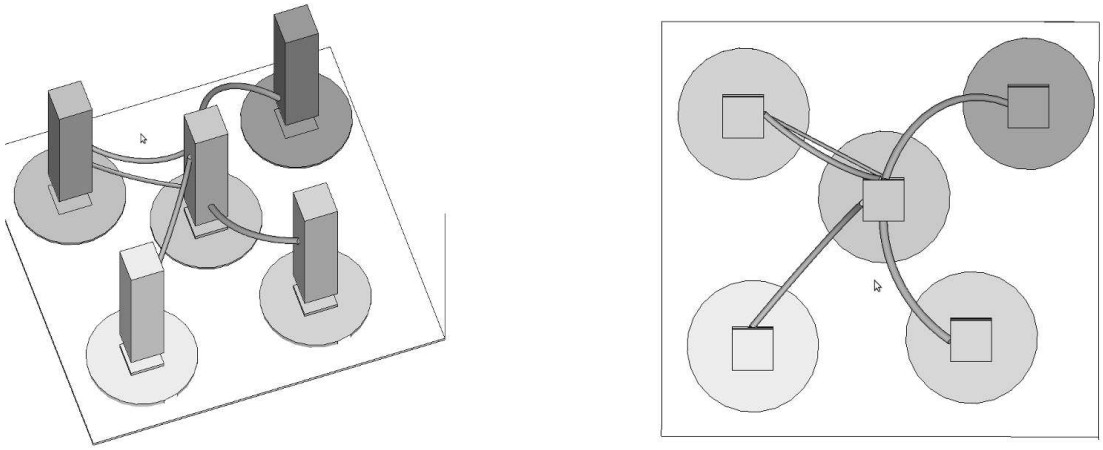
A maioria das ferramentas de visualização não são capazes de efetuar uma análise levando-se em conta a topologia da rede. ParaGraph é a única ferramenta que apresenta uma noção de interconexão em suas técnicas de visualização, embora provendo apenas visualização de hipercubo e padrões de comunicação, separadamente. Na realidade, ParaGraph não foi concebido para a análise de aplicações de larga-escala. Outras técnicas, como espaço-tempo ou baseadas em grafo, usadas em outras ferramentas de visualização, também não são capazes de apresenta a topologia da rede com as comunicações de aplicações paralelas. Neste caso, a limitação é relacionado a forma como os recursos e componentes da aplicação são desenhados, feito em um espaço linear. Quando a plataforma de execução se torna maior e mais complexa, mostrar a topologia da rede em uma visualização espaço-tempo se torna impraticável.

Nossa proposta de fazer uma ligação entre a análise da aplicação e a topologia da rede é baseada em um esquema composto de três dimensões. Uma das dimensões é o tempo, e as outras duas dimensões são usadas para representa a topologia da rede. A próxima seção apresenta a concepção visual do nosso modelo, e a seção seguinte apresenta o modelo de componentes abstratos que pode ser usado para se gerar tal visualização.

A.2.1 Concepção Visual

A concepção visual do nosso modelo consiste na combinação de técnicas de visualização que mostram o comportamento da aplicação com técnicas que mostram dados estruturais ou estatísticos. Se dados estruturais são utilizados, a topologia da rede pode ser usada juntamente com o comportamento da aplicação. Se dados estatísticos são aplicados, o usuário pode simplificar quantitativamente o comportamento da aplicação, em diferentes escalas e fatias de tempo.

O resultado da concepção visual é o modelo tri-dimensional. O modelo tem duas dimensões reservadas para as representações estruturais e estatísticas. Nós nomeamos estas duas dimensões como a base da visualização 3D. A terceira dimensão é a linha do tempo. A Figura A.1(a) mostra um exemplo da abordagem 3D para representação de dados da aplicação. Os estados dos processos são representados como barras verticais que são posicionadas em cima da base da visualização. Os diferentes estados ao longo do eixo do tempo podem ser representados por diferentes cores. Cada representação de estado é colocada verticalmente seguindo suas marcações de início e fim. Comunicações são representadas como flechas ou linhas no ambiente 3D, conectando dois ou mais processos que se comunicam. A Figura A.1(b) mostra a visualização de um diferente ponto de vista, localizado sobre os objetos representados. Esta visão permite a observação do padrão de comunicação da aplicação, por exemplo.



(a) Concepção visual da abordagem 3D. (b) Visualização a partir do topo.

Figura A.1 – Concepção visual da abordagem 3D com rastros de aplicação representados por barras verticais representando o comportamento de processos ao longo do tempo.

A.2.2 Modelo de Componentes

Para criar uma visualização 3D, os rastros coletados das aplicações devem passar por uma série de transformações. Para tal, definimos aqui um modelo de componentes abstratos. A Figura A.2 apresenta a organização geral deste modelo. Como entrada, o modelo usa dois tipos de informação: rastros de aplicações paralelas e um arquivo de configuração contendo a descrição dos recursos do ambiente de execução.

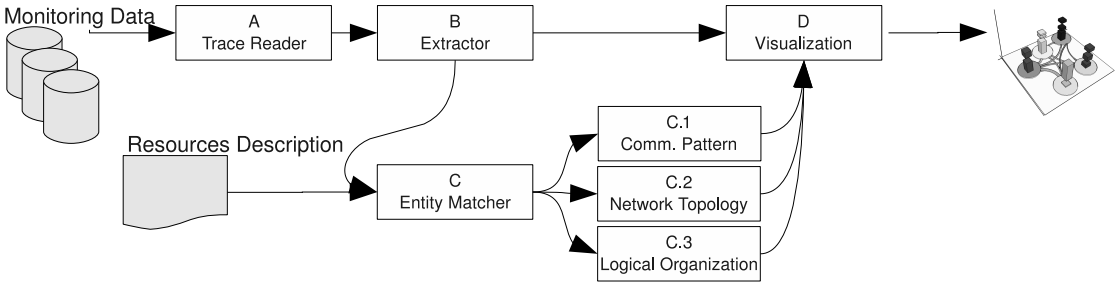


Figura A.2 – Modelo de componentes abstratos da abordagem 3D, com as três configurações possíveis para a base da visualização.

A base da visualização é configurada pelo componente *Entity Matcher* (C). Desenvolvemos três diferentes configurações para o mesmo: uma que mostra o padrão de comunicação da aplicação; outro que mostra este padrão combinado com a topologia da rede; e o último é a combinação dos dados da aplicação com uma representação lógica dos recursos. O componente escolhe uma dessas visualizações de acordo com a escolha do usuário.

Entre as três alternativas modeladas no *Entity Matcher*, a que considerada topologia da rede

(C.2) lida diretamente com o problema da influência da interconexão na aplicação. As outras alternativas são apresentadas para mostrar outras informações estruturais (o padrão de comunicação) e uma representação estatística com detalhes de comportamento ao longo do tempo.

Nós consideramos no modelo que existem arquivos de rastros disponíveis para a leitura, os quais guardam eventos que geram um fluxo que atravessa os componentes da Figura A.2 da esquerda para a direita. Mesmo assumindo arquivos como entrada, os componentes podem funcionar independentes da como os dados de rastreamento são injetados no modelo. Sendo assim, o modelo é capaz de lidar com uma geração online de eventos quando a quantidade dos mesmos não é tão grande. Notificações podem também ocorrer no modelo dos componentes de visualização em direção aos outros componentes, para propagar configurações e mudanças no comportamento iniciadas por comandos de usuário.

A.3 O Modelo Visual de Agregação

Outra questão relacionada a aplicações Grid é que elas podem ser compostas de uma grande quantidade de processos. Algumas análises já são possíveis com grandes aplicações [50], mas somente em clusters. Várias questões surgem em ambientes Grid ao analisar aplicações de larga-escala. Uma primeira é a grande quantidade de dados de monitoramento, que dependem de dois fatores: o número de entidades monitoradas, e a quantidade de detalhe coletada de cada entidade. Outra questão é a escalabilidade visual [26] das técnicas de visualização, que fala sobre a quantidade de dados que podem ser mostrados na tela sem que o usuário perca a habilidade de entender o que é representado.

É fato que as técnicas de visualização das ferramentas devem também ser escaláveis para analisar aplicações paralelas grandes. Se consideramos apenas a quantidade de entidades monitoradas, devemos ser capazes de representar pelo menos alguns milhares de processos na mesma tela. Uma certa quantidade de detalhes também deve estar presente na representação. Um exemplo de técnica de falta de escalabilidade é a representação espaço-tempo, onde a quantidade de dados a ser representada é limitada pelo espaço vertical disponível em telas de computadores.

Entre as ferramentas de visualização existentes, Vampir tem em sua visualização espaço-tempo uma técnica hierárquica que aumenta a quantidade de processos que podem ser visualizados ao mesmo tempo. A técnica funciona através da agregação do comportamento de processos de acordo com a representação hierárquica. O problema da abordagem é que a informação de cada nível é apresentada de forma diferente, tornando difícil a análise de visões agregadas. Outras ferramentas, como Pajé e Jumpshot, usam mecanismos de rolagem para lidar com um número grande de entidades monitoradas. Esta técnica tem um impacto negativo uma vez que o comportamento de todas as entidades não é mostrado ao mesmo tempo.

Nossa abordagem usa intervalos de tempo para criar uma estrutura hierárquica que representa o comportamento da aplicação para o período selecionado. Nós então usamos a técnica Treemap [42] para criar uma representação visual da estrutura. A técnica proposta aumenta a quantidade de entidades que podem ser representadas ao mesmo tempo, e permite uma direta comparação entre as mesmas. Além disso, nós também apresentamos um mecanismo de agregação que pode ser aplicado para mudar a visualização quando existem muitas entidades para ser analisadas na mesma tela. A combinação destas dessas duas técnicas permite se atingir escalabilidade visual na análise de aplicações paralelas.

A.3.1 Algoritmo de Fatia de Tempo

O objetivo do algoritmo de fatia de tempo consiste em criar uma estrutura hierárquica que reflete o comportamento do programa para um dado intervalo de tempo. Para isso, os nós da hierarquia devem receber valores que são calculados baseados em dois fatores: a definição do intervalo de tempo e um sumário de eventos para cada entidade monitorada naquele intervalo. Diferente configurações para definir o intervalo de tempo são possíveis, desde intervalos pequenos até grandes, entre outros.

O sumário de eventos é feito levando-se em conta o intervalo de tempo especificado e informações adicionais sobre uma entidade, presente nos dados de monitoramento. O objetivo é encontrar um valor numérico que represente o comportamento de cada entidade. Existem

diferentes jeitos de definir esses valores numéricos. Podemos considerar que esse número é a quantidade de tempo, ou a quantidade de vezes que algo acontece, ou qualquer outra informação que pode ser contada de algum jeito. O princípio geral do algoritmo é somar separadamente os valores para cada um dos tipos de dados que podem ser encontrados para uma entidade, como estado, variável, links e eventos, e então realizar uma intersecção dessa soma com a fatia de tempo usada.

A.3.2 Agregação Visual

O uso de uma representação Treemap habilita a escalabilidade da análise. Isto significa que se aumentamos o tamanho da hierarquia sendo visualizada, a representação permanece compreensível do ponto de vista do usuário. Embora isto acontece na maioria das situações, a técnica se mantém limitada pelo tamanho do espaço dedicado a sua representação na tela do computador.

O modelo de agregação tenta superar esta limitação através da reorganização da hierarquia a ser visualizada. Ele age basicamente através da agregação de valores das folhas da árvore para nós intermediários da mesma. Com esta abordagem, a renderização Treemap pode ser parada em qualquer nível sem perder a informação importante que foi registrada nos nós folhas da árvore.

Figura A.3 mostra três modificações na hierarquia causadas pelo modelo de agregação. A hierarquia original é mostrada na esquerda. Cada informação nos nós folhas pode representar uma métrica diferente, como a quantidade de vezes que algo acontece. No nosso exemplo, existem três níveis intermediários: Processo (P), Máquina (M) e Cluster (C). O objetivo principal da agregação é agrupar os valores de P e fazê-los subir um nível da árvore. Sendo assim, após a primeira agregação, os valores nos vetores são somados e atribuídos aos nós M. O algoritmo pode ser aplicado novamente para continuar a agregação até o nó raiz.

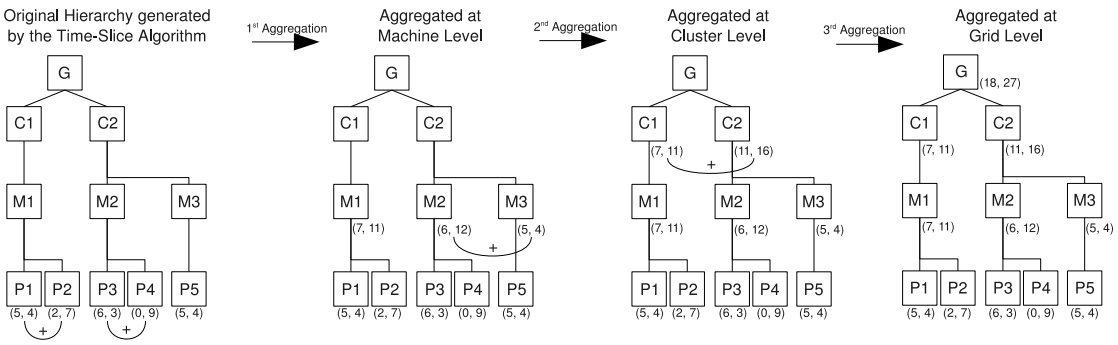


Figura A.3 – Três agregações realizadas pelo modelo de agregação.

Além da tradicional operação de soma (mostrada na Figura A.3, o modelo de agregação pode ser aplicado usando outras funções, como máximos, mínimos, média e mediana. A aplicação dessas funções depende diretamente em qual o tipo de informação sendo agregada e pode ser usado para evidenciar alguma característica particular.

O benefício trazido pelo modelo de agregação é evidente quando o mesmo é combinado com o algoritmo de fatia de tempo. Quando uma aplicação paralela é composta por muitos

processos, a técnica de agregação pode ser usada para melhorar a análise da visualização baseada em treemaps.

A.4 O Protótipo Triva

Esta seção descreve resumidamente o protótipo desenvolvido para implementar os modelos apresentados nas seções anteriores. Esta descrição mostra as decisões de implementação tomadas. O protótipo é chamado de Triva.

Um dos principais guias durante a implementação do protótipo é que ele deveria ser construído sobre ferramentas e bibliotecas existentes, principalmente para evitar a desenvolvimento de implementações já validadas. A primeira decisão tomada é a adoção de algumas partes da ferramenta Pajé. As principais razões que motivaram esta adoção é a reutilização de software e o bom desempenho dos componentes de simulação do Pajé. Outras decisões tomadas incluem o uso de formatos de descrição de recursos facilmente reconhecidos textualmente, a adoção da biblioteca GraphViz, entre outros.

A Figura A.4 mostra a organização geral do protótipo, composta de módulos que transformam os dados de rastreamento em objetos Pajé, e então nos dois tipos de visualização: o 3D e a treemap. Pelo fato da adoção de objetos genéricos, a única parte do protótipo que é dependente do formato do rastro é aquela representada na esquerda da Figura, indicada pelo integrador DIMVisual e seus sub-componentes. Os retângulos brancos são bibliotecas e ferramentas existentes que foram reutilizadas com poucas alterações; retângulos cinzas foram desenvolvidos para fazerem parte do protótipo.

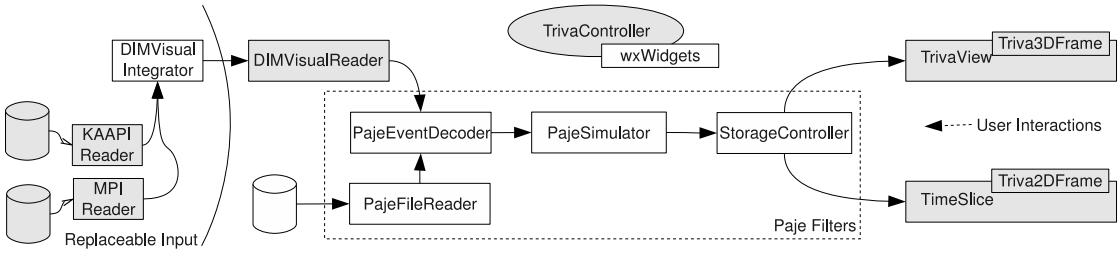


Figura A.4 – Arquitetura Triva.

O componente TrivaController, escrito na linguagem C++, fica a cargo da inicialização de todos os componentes, conectando-os seguindo a arquitetura da Figura A.4. Ele também apresenta ao usuário a interface gráfica, criada usando a biblioteca wxWidgets, através de uma janela, com opções de configuração e mecanismos de interação. A cena tri-dimensional e a renderização treemap é também inicialmente configurada por esse componente.

Os filtros Pajé, representados pelos retângulos pontilhados da Figura A.4, são os mesmos utilizados na ferramenta de visualização Pajé. Suas implementações levam em conta várias questões ligadas a escalabilidade e baixo tempo de resposta aos comandos da interface com o usuário. O primeiro dos filtros, PajeEventDecoder, lida com a entrada gerada pelo DIMVisualReader e prepara para o próximo módulo. O PajeSimulator transforma os eventos em objetos visuais. Esta transformação consiste em criar uma estrutura hierárquica dos rastros, usando os tipos básicos Pajé. Esta estrutura, que representa a mesma informação encontrada nos rastros, é otimizada para a visualização, e registrada no StorageController.

Na parte mais da direita da Figura A.4, as interações entre os módulos funcionam nos dois

sentidos. Interações da direita para a esquerda são pedidos de novos dados. Eles são lançados por comandos de usuário ou mudanças nas configurações. As interações da esquerda para a direita são respostas aos pedidos.

A.4.1 TrivaView

O modelo de visualização, apresentado na seção A.2, é implementado no protótipo Triva através do componente TrivaView. A Figura A.5 apresenta sua organização geral incluindo os componentes relacionados. O módulo TrivaView implementa a parte do Extractor do modelo 3D, obtendo do fluxo de objetos Pajé os containers e links, e redirecionando o fluxo para o componente DrawManager. A parte do modelo 3D que se chama Entity Matcher é implementada em três components do protótipo: TrivaApplicationGraph, TrivaResourcesGraph and TrivaTreemapSquarified. Eles recebem como entrada os containers e links do TrivaView, e a descrição dos recursos de arquivo. A parte Visualization do modelo 3D mostrada através do cículo pontilhado na direita da Figura A.5, é implementada com 4 componentes: o Triva3DFrame, que mantém a cena 3D, e seus três gerenciadores que podem mudar os aspectos visuais, o DrawManager, o AmbientManager e o CameraManager.

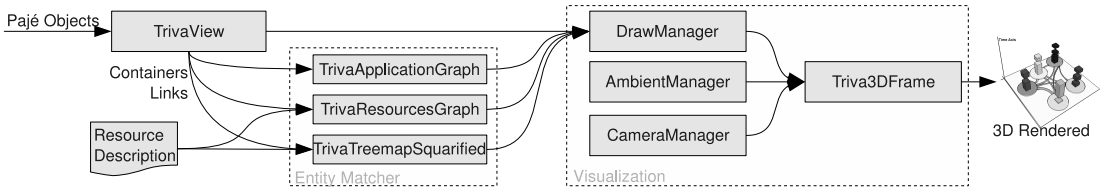


Figura A.5 – Layout de implementação do TrivaView.

A.4.2 TimeSliceView

O modelo de agregação e o algoritmo de fatia de tempo foram implementados no componente TimeSliceView, como mostrado na Figura A.6. Outro componente importante desta Figura é o Triva2DFrame, cuja responsabilidade é desenhar a treemap na janela de visualização do protótipo.

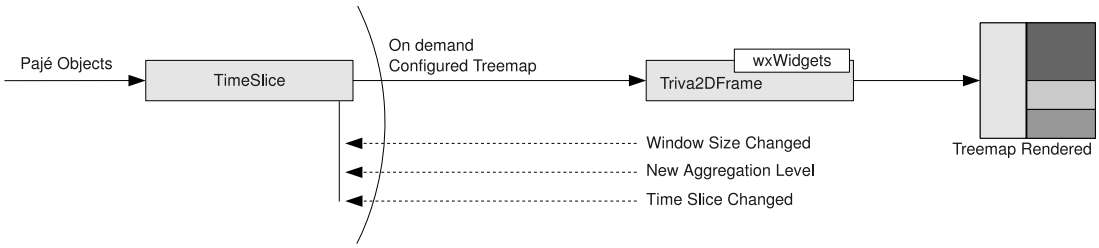


Figura A.6 – Layout de implementação do TimeSliceView.

A Figura A.6 também detalha as interações e notificações que acontecem durante a execução do componente. A chegada de objetos do simulador Pajé (veja Figura A.4 para detalhes) é representada na esquerda da Figura. As interações com o usuário podem causar três diferentes tipos de notificações que vão do componente Triva2DFrame para o TimeSlice: a mudança do tamanho da janela, um novo nível de agregação e a mudança da fatia de tempo. Todas estas notificações disparam a mesma cadeia de acontecimentos no componente: criação de uma hierarquia de comportamento, possível aplicação de operadores de agregação e cálculo da nova treemap. A treemap resultante é enviada como resposta e então desenha na janela pelo componente Triva2DFrame.

A.5 Resultados e Avaliação

O principal objetivo desta parte do resumo é mostrar os dois tipos de visualizações gerados pelo protótipo Triva, um deles tri-dimensional e outro com treemap. Em paralelo a essas visualizações, é feita uma análise considerando os rastros de execução utilizados como entrada para o protótipo.

A.5.1 Tri-Dimensional

O principal objetivo do modelo tri-dimensional é realizar o mapeamento dos componentes da aplicação com a topologia de interconexão dos recursos. Para apresentar um dos resultados obtidos com este tipo de visualização, selecionamos um cenário onde existem 60 processos, alocados em 2 sites diferentes do Grid'5000. O site *nancy* contribui para a execução com 30 máquinas do cluster *grelon*, ao mesmo tempo que o site *rennes* tem uma alocação de 25 máquinas do cluster *paramount* e 5 máquinas do cluster *paraquad*. Consideramos neste caso que uma topologia de rede no qual cada site contém um roteador próprio e todos os clusters de um site são conectados no seu respectivo roteador. Os roteadores de sites diferentes estão interconectados. Sendo assim, quando uma mensagem é enviado de um cluster de um site a um cluster de outro site, ela deve passar através dos dois roteadores.

A Figura A.7 mostra duas capturas de tela do protótipo Triva geradas durante a visualização do arquivo de rastro deste cenário. O texto e as linhas pontilhadas foram manualmente inseridas para aumentar o entendimento do exemplo. A imagem **A** desta Figura mostra o tempo total de execução com uma escala de tempo pequena, fazendo com que todos os objetos fiquem perto da base da visualização. A linha pontilhada desta imagem mostra a separação entre os sites *rennes*, com dois clusters, e *nancy*, com apenas um cluster. Nós podemos observar nesta escala de tempo que um grande número de roubo de tarefas acontece entre os clusters *grelon* e *paraquad*, provavelmente devido ao maior número de processos alocados neles. Analisando essas interações com a topologia da rede, o protótipo Triva permite que o usuário visualize que todos os pedidos de tarefas destes clusters devem ser comunicados através dos dois roteadores da interconexão.

O protótipo também permite a mudança dinâmica da escala do tempo, usando o mouse. A imagem **B** da Figura A.7 mostra o tempo total de execução para os rastros deste cenário, com uma maior escala de tempo. Através desta imagem, é possível observar as diferenças do comportamento do roubo de tarefas em diferentes intervalos de tempo da execução. Pode-se perceber que no início há um número significativamente menor de roubos comparado com o fim. Isto ocorre porque no fim de uma aplicação KAAPI as tarefas disponíveis para execução se tornam mais raras. Este comportamento é esperado na atual implementação do KAAPI, onde um roubo de tarefas aleatório é implementado.

Um segundo cenário é uma aplicação KAAPI composta por 200 processos, em 200 máquinas. A alocação de máquinas está dividida em dois sites: *rennes* e *nancy*. O número de máquinas alocadas em cada um é igual, embora a alocação interna de cada um difere em quantidade de máquinas por cluster. A imagem **A** da Figura A.8 mostra o número de máquinas para cada cluster alocado e também a topologia da rede que interconecta os dois sites. A linha pontilhada é utilizada para separar os sites. Nós consideramos para este cenário informações adicionais

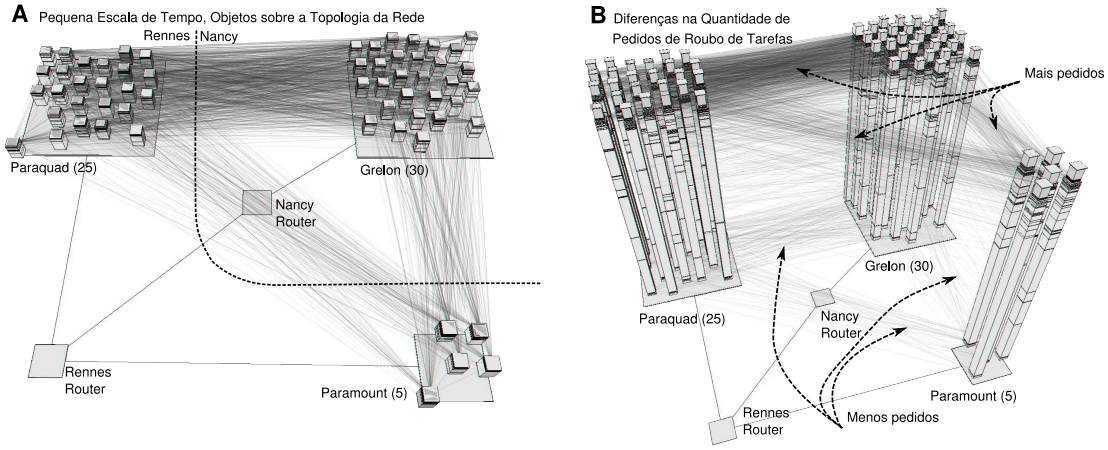


Figura A.7 – Duas capturas de tela do protótipo Triva durante a visualização de uma aplicação composta de 60 processes, em diferentes escalas de tempo.

relacionadas a interconexão entre os roteadores e os três clusters. A largura de banda disponível entre os clusters *paravent* e *grillon*, através dos dois roteadores, é de 100 megabits. O link entre o cluster *grelon* e seu roteador é de 1 megabit, como mostrado na imagem **A** da Figura.

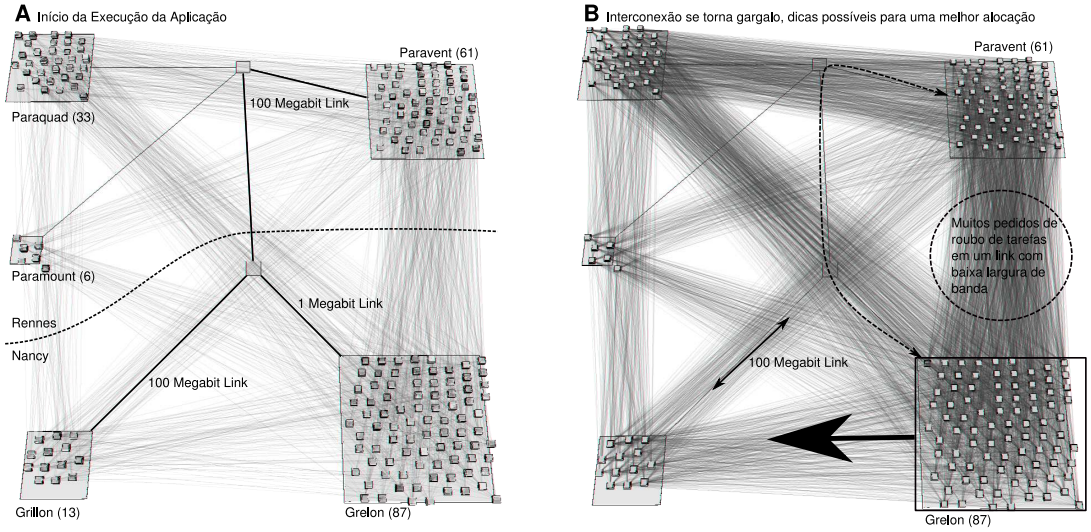


Figura A.8 – Duas visões de um exemplo com mais informações da topologia da rede, como as limitações impostas pela largura de banda.

Neste cenário, existem 87 processos executando no cluster *grelon*, e 61 no *paravent*. Considerando apenas os roubos de tarefas entre estes dois clusters, como mostrado no círculo pontilhado da imagem à direita da Figura A.8. A flecha pontilhada da mesma imagem indica que estes pedidos devem passar através do link de 1 megabit. A visualização sugere que um número

menor de processos deveria ser colocado em um cluster com largura de banda limitada. Se, por exemplo, os processos do cluster *grelon* fossem executados no cluster *grillon*, a execução poderia obter um melhor desempenho.

Através do exemplo deste segundo cenário, nós podemos notar a importância da análise do desempenho de uma aplicação juntamente com uma representação topológica da rede. Se este tipo de visualização, como mostrado na imagem **B** da Figura A.8, não estiver presente, o analista pode obter conclusões erradas sobre o desempenho da aplicação.

A.5.2 Agregação

Um dos principais benefícios do modelo de agregação de dados desta tese é a facilidade de análise uma grande quantidade de entidades monitoradas na mesma tela. Para avaliar quão escalável é a visualização, nós geramos um rastro sintético composto de 100 mil processadores, cada um com dois estados diferentes. Segue a seguir a análise desse rastro com a técnica de fatia de tempo e o algoritmo de agregação.

A Figura A.9 mostra a análise do rastro, cuja hierarquia tem quatro níveis: Site, Cluster, Machine e Processor. A hierarquia tem 10 Sites, cada qual com 10 Clusters, cada cluster com 100 Machines e cada machine com 100 processors. Cada processador pode estar em um de dois estados possíveis, representados na Figura pelas tonalidades fraca e forte de cinza.

A análise em larga-escala usando o protótipo começa com a treemap **A**, localizada no topo à esquerda da Figura A.9, no nível processor. Nesta treemap, existem 200 mil retângulos: 100 mil processadores vezes a quantidade de estados possíveis, que são 2. Nós podemos observar que algumas regiões desta treemap são mais escuras que outras, permitindo algum tipo de conclusão. Entretanto, qualquer conclusão precisa é difícil de obter com esta treemap. A principal razão disso é que a treemap **A** tem retângulos que são muito pequenos, tornando difícil a observação de diferenças de tamanho entre dois estados de um único processador. O exemplo é mostrado para indicar a limitação de uma visualização treemap tradicional.

O retângulo branco da treemap **A** na Figura A.9 representa o espaço dedicado para uma máquina. Embora seja difícil de notar, existem 200 retângulos nesta pequena área que representam o estado dos 100 processadores desta máquina. Pelo fato de ser difícil de entender o padrão de todos esses 100 processadores, o usuário pode interagir com o protótipo e mostrar valores agregados para o nível máquina, como mostrado na treemap **B** da Figura. Ela mostra para cada máquina os dois possíveis estados. Nesta visão, já é possível analisar diferenças entre as máquinas: algumas estão significativamente mais em um estado do que em outro. A área em evidência no lado esquerdo da treemap **B**, mostrada através de um zoom, corresponde a área do retângulo branco da treemap **A**.

As agregações seguintes permitem o usuário de visualizar os rastros no nível de cluster, como mostrado na treemap **C** da mesma Figura, e no nível de site na treemap **D**. A treemap C mostra 100 clusters (10 por site). Em seu lado esquerdo, a treemap apresenta um retângulo preto que mostra 10 clusters na área dedicada para um site. A flecha começando neste retângulo aponta para os valores agregados para este site, na treemap D. A máxima agregação possível, mostrada na treemap **E**, permite uma visão por estado das informações disponíveis, indicando que o estado representado pela tonalidade mais clara aparece mais vezes que o outro na fatia de tempo selecionado para este exemplo.

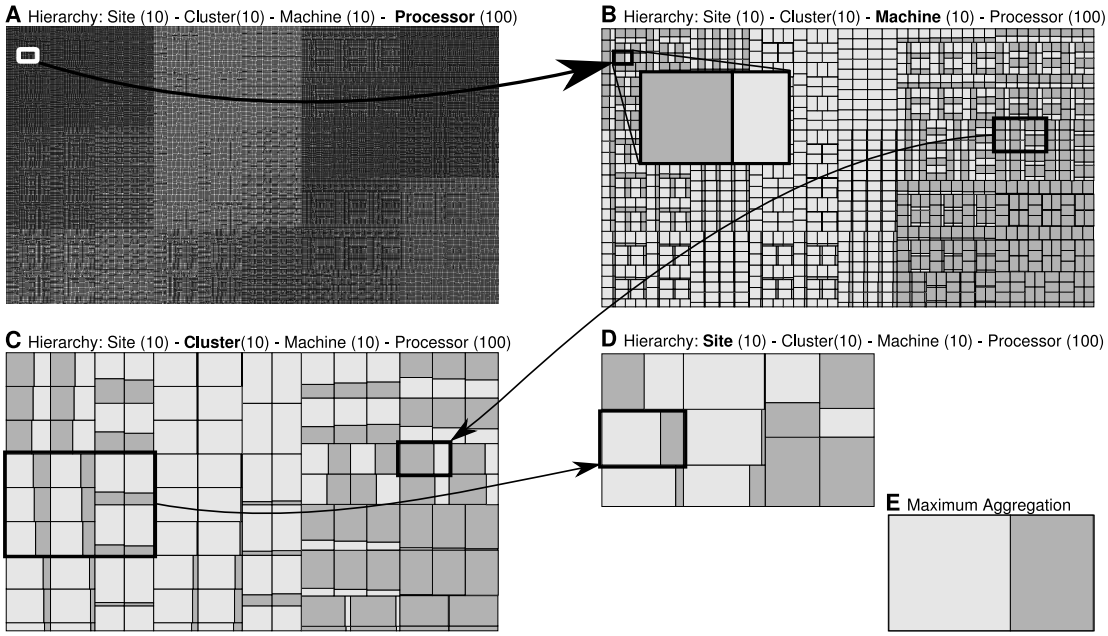


Figura A.9 – Treemap Normal (A) e quatro treemaps com dados agregados (B – E) de dois estados para 100 mil processadores (gerados sinteticamente).

Um segundo cenário para a visualização treemap é uma aplicação de 188 processos, executada em 188 máquinas, distribuídas em cinco sites do Grid'5000 incluindo o cluster de Porto Alegre. Existem 13 máquinas alocadas do cluster *xiru*, em *portoalegre*; 2 de *bordereau*, 17 de *bordemer*, e 6 de *bordeplage*, em *bordeaux*; 45 de *pastel*, 5 de *violette*, em *toulouse*; 14 de *paramount*, 36 de *paraquad*, em *rennes*; e finalmente 50 de *grelon* no site *nancy*. A Figura A.10 mostra duas treemaps calculadas com os rastros gerados neste cenário.

A treemap **A** mostra os estados Run e RSteal para todos os 188 processos. Quase todos os processos mostram o mesmo comportamento, com o estado Run maior (áreas com ton cinza claro) comparado com o estado RSteal (cinza escuro). A única exceção aparece nos K-processos executados no site de *portoalegre*, colocados em evidência manualmente com o círculo pontilhado. Observando esta treemap, nós notamos que estes processos passam mais tempo roubando tarefas que os processos de outros sites. A treemap **B**, na direita, mostra a mesma fatia de tempo e os mesmos processos, mas somente o estado RSteal. Aqui, a diferença de tempo despendida roubando tarefas se torna ainda mais evidente. Nós acreditamos que a principal razão atrás deste comportamento vem da interconexão entre os sites. O site de *portoalegre* é localizado no Brasil, e a sua conexão com o Grid'5000 é feita través de uma Rede Privada Virtual (VPN) que é mantida através da internet. A latência desta interconexão, comparada com a latência geral entre os sites do Grid'5000 localizados na França, é significativa. O roubo de tarefas tradicional implementado no KAAPI não diferencia quem será o alvo do roubo. Isto, em um ambiente de interconexão heterogêneo, pode levar a mais tempo gasto para roubar, como indicado através da treemap calculada através do nosso algoritmo de fatia de tempo.

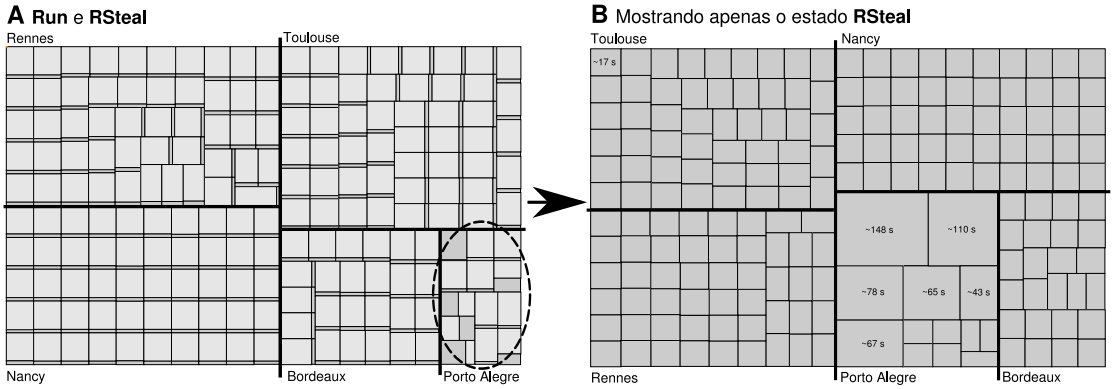


Figura A.10 – Cenário KAAPI com uma aplicação composta de 188 processos.

No geral, o algoritmo de fatia de tempo combinado com o modelo de agregação desta tese possibilita uma fácil identificação de questões de desempenho ao comparar o comportamento de processos de uma aplicação paralela. O modelo de agregação aporta vantagens para situações de larga-escala, não importanto quantos processos estão envolvidos na análise. O único passo necessário para as duas propostas funcionarem bem nestes ambientes é a definição de uma hierarquia com ao menos alguns níveis. As hierarquias usadas neste cenário KAAPI tem 5 níveis, tornando possível a obtenção de bons resultados na visualização.

A.6 Conclusão e Trabalhos Futuros

Esquemas de visualização tradicionais para análise de aplicações paralelas foram concebidos para lidar com dados de monitoramento de pequena escala e de ambientes equilibrados. A necessidade de técnicas de visualização para a análise de aplicações para ambientes de larga-escala, tais como Grids, motiva este trabalho. Dois problemas na análise de aplicações paralelas através da visualização são identificados nesta tese.

O primeiro é o impacto da rede de interconexão na execução de aplicações paralelas. Este impacto deve estar presente na análise para se melhor entender e melhorar o desempenho da aplicação. Técnicas de visualização tradicionais, como a representação espaço-tempo por exemplo, são largamento usadas para análise de aplicações. No entanto, estas técnicas não conseguem mostrar na mesma tela a topologia da rede e os dados de monitoramento da aplicação. Isto pode levar a conclusões erradas durante a detecção de problemas de desempenho das aplicações. O segundo problema é a escalabilidade visual das técnicas de visualização. Normalmente, o número de entidades monitoradas que pode ser analisado na mesma tela é limitado à resolução vertical da tela de um computador. Representações espaço-tempo são um claro exemplo deste problema, não sendo bem apropriadas para a análise de aplicações Grid compostas por um número grande de processos.

A idéia principal desta tese é a exploração de técnicas de visualização da informação que podem ser utilizadas para analisar o comportamento de aplicações paralelas. No nosso caso, esta exploração também considera os dois problemas que tentamos resolver. Nossa primeira abordagem mostra a rede de interconexão juntamente com os dados da aplicação usando uma visualização tri-dimensional, onde a base desta visualização é usada para detalhar a interconexão entre os recursos, e o terceiro eixo para mostrar a evolução da aplicação ao longo do tempo. Nós melhorarmos nossa solução através da representação de padrões de comunicação, oferecendo ao desenvolvedor a possibilidade de casar este padrão com o da topologia da rede.

A segunda abordagem é o modelo de agregação visual, onde os problemas de escalabilidade são superados através da combinação da técnica Treemap e o algoritmo de fatia de tempo. Este algoritmo leva em conta intervalos de tempo para gerar valores e injetá-los em uma organização hierárquica da aplicação. Esta estrutura é então representada através da técnica Treemap. A escalabilidade da visualização é atingida através do modelo de agregação, onde os níveis da hierarquia são explorados para criar dados intermediários que pode ser usados para criar visualizações treemap por níveis com mais informações.

Ambas as abordagens foram implementadas em um protótipo chamado Triva, desenvolvido usando um gerenciador de cena 3D chamado Ogre e uma implementação de Treemap própria. O protótipo tem mecanismos de leitura que o ligam com a biblioteca de integração DIMVisual, capaz de integrar dados de monitoramento de diferentes fontes e formatos. Rastros sintéticos e reais do KAAPI e MPI foram usados para validar as abordagens e a implementação. Os rastros KAAPI foram coletados na plataforma Grid'5000. Embora a avaliação do protótipo é ligada aos rastros usados, o uso do formato Pajé como entrada permite a extensão dos benefícios da ferramenta para outros campos de pesquisa e aplicação, de visualização de recursos a outros tipos de bibliotecas de comunicação.

Os resultados obtidos são promissores. A visualização tri-dimensional permite um melhor

entendimento de padrões de comunicação com a topologia da rede. Nós usamos uma simplificação da topologia do Grid'5000 e o roubo de tarefas de aplicações KAAPI. Fomos capazes de mostrar que em diferentes fatias de tempo, o roubo de tarefas poderia se beneficiar mais da localidade, uma vez que a implementação atual de KAAPI não leva em conta dados da rede para realizar pedidos de roubo de tarefas. Por outro lado, os resultados obtidos com o modelo de agregação permitiram a visualização dos estados de 100 mil processadores, gerados sinteticamente. As treemaps definidas pelo algoritmo de fatia de tempo foram também definidas usando rastros reais KAAPI e MPI. Fomos capazes de identificar nos rastros KAAPI diferentes características, como o comportamento diferente nos mecanismos de roubo apresentados por diferentes processos, a eficiência do balanceamento de carga considerando todo o tempo de execução das aplicações, e a análise em larga-escala de uma aplicação KAAPI composta por quase 3 mil processos.

Em resumo, os principais objetivos alcançados nesta tese são a proposta da abordagem 3D, o modelo de agregação visual combinado com o técnica de fatia de tempo e protótipo Triva. Além disso, se incluem a interação entre o protótipo Triva e a biblioteca KAAPI, permitindo uma análise das atividades de roubo de tarefas desta biblioteca.

Como perspectivas de trabalhos futuros, prevê-se a evolução da visualização 3D para a representação de informações geradas pelo modelo de agregação; criação de representações de grafo com a técnica de fatia de tempo e agregação; o estudo de outras funções de agregação e outros tipos de dados para o algoritmo de fatia de tempo. Acreditamos que a implicação mais significativa deste trabalho seja o estudo de técnicas de visualização aplicadas para a análise de aplicações paralelas.

Appendix B

Extended Abstract in French

The French title for this thesis is *"Quelques Modéles de Visualisation pour l'Analyse des Applications Paralléles"*. The extended french abstract is also presented here to fulfill the requirements established in the *co-tutelle* agreement of the author. This abstract is a french translation of previous Portuguese extended abstract.

B.1 Introduction

Les systèmes distribués sont fondés sur du matériel et des logiciels contenant et gérant plus d'une unité d'exécution [19]. Dans ces systèmes, les processeurs sont interconnectés et communiquent via un réseau. Les programmes pour ces machines sont divisées en plusieurs catégories et doivent interagir à différents niveaux de parallélisme, tels que le passage de messages ou la mémoire partagée. Un exemple de système distribué est représenté par les grilles de calcul [30]. Ce type de système est structuré en organisations virtuelles [29], et peut-être composé de milliers de machines distribuées géographiquement. Deux exemples de grilles sont le projet français Grid'5000 [12] et le projet américain TeraGrid [16].

Les caractéristiques partagées par presque toutes les plates-formes de type grille sont le dynamisme, l'hétérogénéité des ressources et des logiciels et la présence de multiples domaines administratifs. Le dynamisme signifie que les ressources d'une grille peuvent devenir indisponibles à tout moment, sans aucune notification préalable. Les applications parallèles doivent considérer ces conditions dynamiques typiquement pour faire face aux fluctuations de la quantité de ressources disponible. L'hétérogénéité signifie que différentes configurations de ressources sont présentes sur la même plate-forme de grille. Ceci est également valable pour les logiciels de bibliothèques. Une grille peut être composée par les différents domaines administratifs, où chaque partie est maintenue indépendamment par leur administrateurs. Au-delà de ces caractéristiques, une grille peut également être connectée par un réseau complexe et être facilement étendue par l'ajout de nouvelles ressources.

L'interconnexion entre les ressources d'un réseau peut être composée de différents types de réseau : Ethernet, Myrinet, InfiniBand, ou fibre optique. Un exemple de grille contenant plusieurs types d'interconnexion est appelé *Desktop Grids* [48], comme les projets BOINC [1] et Seti@Home [2], où l'interconnexion se fait généralement par le biais d'Internet. Autre exemple qui montre la présence de plusieurs types d'interconnexions est une grille composée de *clusters*, où une forte hiérarchie d'interconnexion est utilisée pour connecter des *clusters* homogènes [12]. La présence de plusieurs types d'interconnexion est un reflet de l'hétérogénéité et la répartition géographique de grilles. Ces aspects imposent un réseau plus complexe, un nombre plus grand de directives de routage pour la communication entre les processus et une latence variable dans le temps.

Les plate-formes de type grille passent facilement à l'échelle car de nouvelles ressources peuvent y être ajoutées indéfiniment en les reliant aux participants existants. En règle générale, ces compléments apportent plus d'hétérogénéité et de complexité au niveau de réseau. Actuellement, il existe des grilles globales composées de milliers d'ordinateurs, comme le montre l'exemple du projet BOINC. Un autre exemple qui montre comme il est facile d'ajouter de nouvelles ressources à une grille est Grid'5000, où de nouveaux *clusters* sont ajoutés au *backbone* principal de la plate-forme. Le passage à l'échelle de ces plate-formes est une bonne chose pour les applications parallèles, qui exigent de plus en plus de ressources informatiques.

Toutes ces caractéristiques de la grille influencent directement le comportement des applications parallèles au cours de leur développement et leur mise en exécution. De ce fait, il est important que le développeur comprenne les impacts des systèmes distribués sur l'application. L'analyse d'une application parallèle qui depend de la topologie du réseau est un exemple. L'ap-

plication peut avoir un performance qui varie en fonction des ressources qui ont été sélectionnées et l'interconnexion entre elles. Cette influence est encore plus évidente lorsque les caractéristiques de réseau sont considérées, comme la latence et la bande passante, pour les applications qui sont limitées par celui-ci. Le passage à l'échelle d'une grille est un autre aspect qui influence directement le comportement des applications parallèles, la disponibilité de nouvelles ressources pour l'application ne signifie pas que l'exécution aura une meilleure performance.

Compte tenu de ces éléments, nous pouvons voir qu'il est important d'analyser le comportement des applications parallèles en conjonction avec les informations de la grille. Cette analyse peut aider les développeurs à comprendre l'impact de la topologie du réseau sur l'application, par exemple. En visualisant la façon dont l'application communique et la topologie du réseau, il est possible de determiner comment l'adapter afin de mieux exploiter cette interconnexion. En outre, si le réseau est hiérarchiquement organisé, les applications peuvent suivre sa hiérarchie pour éviter les goulets d'étranglement. Une bonne analyse doit aussi conduire à des conclusions sur tous les processus qui sont mis en exécution, y compris sur les comportements locaux et globaux qui peuvent apparaître entre eux. Quand il y a une grande quantité de processus, l'analyse doit être en mesure de générer des résultats statistiques sur l'ensemble de ces processus.

La visualisation est une forme d'aide à l'analyse des applications parallèles. Elle a été largement utilisé au cours des 30 dernières années, pour comprendre et visualiser les applications qui sont axées sur différents niveaux de parallélisme. La façon la plus classique de construire une visualisation consiste à utiliser une adaptation des diagrammes de Gantt [79], également connue sous le nom de graphiques d'espace-temps. Ces visualisations disposent la liste des composants de l'application verticalement et metent la ligne du temps sur l'axe horizontal. Des exemples d'outils qui offrent ce type d'analyse sont l'outil de visualisation générique Pajé [22], Vampir [60] et d'autres [5, 46, 63]. Ces graphiques espace-temps sont déjà largement utilisés dans les plates-formes existantes, tels que les *clusters*, où les données sont simples et uniformes.

Beaucoup de ces outils de visualisation ont été adaptés afin d'observer le comportement des applications dans les systèmes distribués, comme les grilles. Habituellement, ils continuent à utiliser les même techniques de visualisation. Considérant les représentations espace-temps, le premier problème qui se pose est qu'elles ne peuvent pas représenter, avec les données de l'application, la complexité de la topologie du réseau d'une grille. Comme nous l'avons dit, l'impact de la la topologie ne peut pas être exclu de l'analyse quand l'interconnexion entre les ressources est complexe. Le deuxième problème est lié au passage à l'échelle de l'affichage graphique espace-temps. Avec l'utilisation de ces représentations, le nombre de composantes de l'application qui peuvent être visualisés dans un écran d'ordinateur est limité à la résolution verticale de l'écran.

Cette thèse tente de résoudre les problèmes des techniques traditionnelles dans la visualisation des applications parallèles. L'idée principale est d'exploiter le domaine de la visualisation de l'information et essayer d'appliquer ses concepts dans le cadre de l'analyse des programmes parallèles. Portant de cette idée, la thèse propose deux modèles de visualisation : les trois dimensions et le modèle d'agrégation visuelle. Le premier peut être utilisé pour analyser les programmes parallèles en tenant compte de la topologie du réseau. L'affichage lui-même se compose de trois dimensions, où deux sont utilisés pour indiquer la topologie et la troisième est utilisée pour représenter le temps. Le second modèle peut être utilisé pour analyser des applica-

tions parallèles comportant un très grand nombre de processsus. Ce deuxième modèle exploite une organisation hiérarchique des données utilisée par une technique appelée Treemap pour représenter visuellement la hiérarchie. Les deux modèles constituent une nouvelle façon d'analyser visuellement les applications parallèles , car ils ont été conçus pour les systèmes distribués grands et complexes, tels que les grilles.

Quelques concepts proposés dans cette thèse ont été publiés et un article est en cours d'évaluation.

Ce résumé étendu est organisé en cinq sections, de la façon suivante :

Section B.2 : Le Modèle Tridimensionnel

Cette section présente le premier modèle de cette thèse, constitué par l'approche en trois dimensions. Nous décrivons la conception visuelle et une organisation générale de composants pour la génération de visualisations 3D.

Section B.3 : Le Modèle d'agrégation des Données

La section présente l'algorithme de tranche de temps pour la description du comportement d'une application sous forme d'une hiérarchie, et le modèle d'agrégation utilisé pour atteindre le passage à l'échelle dans la représentations *Treemap*.

Section B.4 : L'implementation du Prototype Triva

Cette section présente le prototype développé pour cette thèse. Sa description dans cette partie comprend l'organisation générale de ses composants.

Section B.5 : Résultats obtenus et Évaluation

Les résultats obtenus avec le prototype Triva sont présentés dans cette section. Deux études de cas y sont presentes : une par rapport au modèle tridimensionnel, l'autre liée au modèle d'agrégation visuelle.

Section B.6 : Conclusion

Les résultats et implications de la thèse sont présentés, ainsi que les perspectives pour les travaux futurs.

B.2 Le Modèle Tridimensionnel

La performance des applications parallèles exécutées sur une grille est liée aux caractéristiques de l'interconnexion du réseau [49]. Quand les ressources ont une forte hiérarchie entre elles, le choix de celles assignées à une application sera décisif pour sa performance mais aussi pour sa compréhension. Sans information sur la topologie du réseau, l'analyste n'est pas en mesure de voir que les problèmes sont dus à la mise en oeuvre des communications. Les décisions prises à partir d'une vision traditionnelle dans ce cas peuvent conduire à des conclusions erronées sur la performance. Ainsi, si nous avons été en mesure d'examiner l'exécution en tenant compte des caractéristiques du réseau, nous pouvons voir plus clairement les raisons du comportement de l'application.

La plupart des outils de visualisation ne sont pas en mesure d'effectuer une analyse en tenant compte de la topologie du réseau. ParaGraph est le seul outil qui offre un concept de l'interconnexion dans ses techniques de visualisation, mais seulement par l'affichage séparés de l'hypercube et des modes de communication. En effet, ParaGraph n'a pas été conçu pour l'analyse des applications à grande échelle. D'autres techniques telles que le graphique espace-temps, utilisé dans d'autres outils de visualisation, ne sont pas capables de présenter la topologie du réseau de communication des applications parallèles. Dans ce cas, la limitation est liée à la façon dont les ressources et les composants de l'application sont représentés dans un espace linéaire. Lorsque la plate-forme d'exécution devient de plus en plus complexe, montrer la topologie du réseau dans un affichage espace-temps devient impraticable.

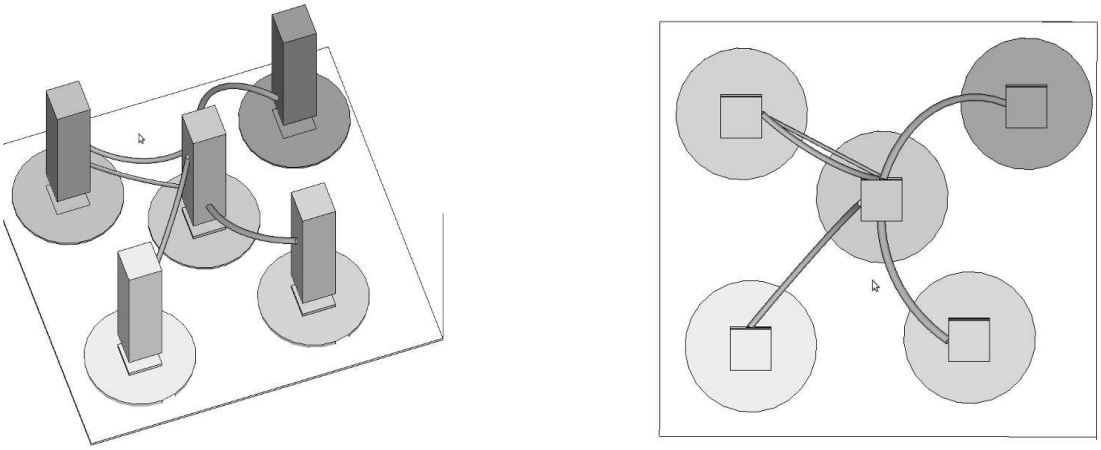
Notre proposition d'établir une connexion entre l'analyse de l'application et la topologie du réseau est fondée sur un système composé de trois dimensions. Une des dimensions est la ligne du temps, et les deux autres dimensions sont utilisés pour représenter la topologie du réseau. La prochaine section présente la conception visuelle de notre modèle, et la section suivante présente le modèle abstrait de composants qui peut être utilisé pour produire ce resultat.

B.2.1 Conception Visuelle

La conception visuelle de notre modèle est composée par la combinaison de techniques de visualisation qui montrent le comportement de l'application avec les données structurelles ou statistiques de celle-ci. Si les données structurelles sont choisies, la topologie du réseau peut être utilisée avec le comportement de l'application. Si les données statistiques sont requises, l'utilisateur peut simplifier quantitativement les données à tracer, à des échelles et des tranches de temps différentes.

Le résultat de la conception visuelle est le modèle tridimensionnel. Le modèle a deux dimensions réservés pour la représentation des données statistiques ou structurelles. Nous avons nommé ces deux dimensions la "base de la visualisation 3D". La troisième dimension est la ligne de temps. La Figure B.1(a) montre un exemple de l'approche par représentation en 3D avec les données d'une application. Les états des processus sont représentés par des barres verticales qui sont placées au-dessus de la base. Les différents états le long de l'axe du temps peut être représenté par des couleurs différentes. La représentation de chaque état est placée verticalement selon ses marques de début et de fin. Les communications sont représentées par des flèches ou des lignes dans un environnement 3D en reliant deux ou plusieurs processus qui communiquent.

Figure B.1(b) montre un point de vue différent, situé au dessus des objets représentés. Ce point de vue permet l'observation de la structure de la communication de l'application, par exemple.



(a) Conception visuelle de l'approche 3D. (b) Diffèrent point de vue.

FIG. B.1 – La conception visuelle de l'approche 3D avec les traces d'une application représentées par des barres verticales montrant l'évolution des processus dans le temps.

B.2.2 Modèle de Composants

Pour créer un affichage 3D, les traces collectées lors de l'execution des applications passent à travers une série de transformations. À cette fin, nous proposons ici un modèle abstrait de composants. Figure B.2 montre l'organisation globale de ce modèle. En entrée, le modèle utilise deux types d'informations : des traces d'applications parallèles et un fichier de configuration contenant la description des ressources de l'environnement d'exécution.

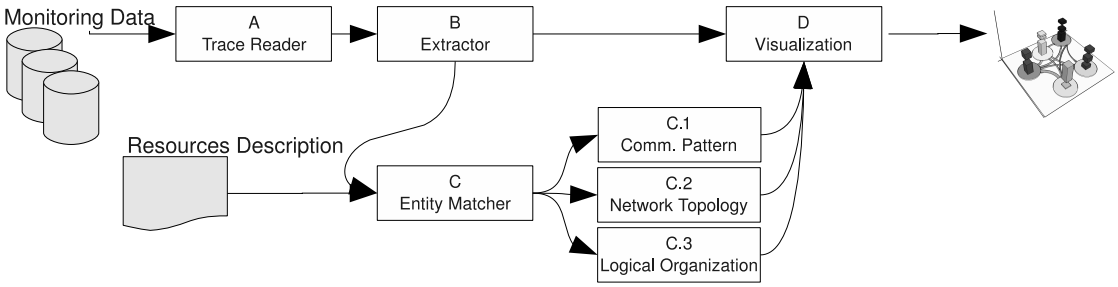


FIG. B.2 – Modèle abstrait de composants pour l'approche 3D, avec trois configurations possibles pour la base de la visualisation.

La base de la visualisation est configuré par le *Entity Matcher* (C). Nous avons développé trois configurations différentes pour celui-ci : celle qui montre le schéma de communication de l'application, celle qui montre ce modèle combiné avec la topologie du réseau, et la denière qui

combine les données provenant de l'application avec une représentation logique des ressources. Le composant selectionne une de ces configurations selon le choix de l'utilisateur.

Parmi les trois alternatives du *Entity Matcher*, celle qui considère la topologie du réseau (C.2) traite directement le problème de l'influence de l'interconnexion sur l'application. Les autres variantes sont présentées pour montrer d'autres informations, comme les donnés structurelles (le modèle decommunication de l'application) et une représentation statistique des détails de son comportement au fil du temps.

Nous considérons dans le modèle l'existence des traces qui sont donc disponibles pour la lecture et qui sont transformées en un flot qui traverse les éléments de la Figure B.2 de gauche à droite. Même en supposant l'existence de ces fichiers d'entrée, les composants peuvent fonctionner indépendamment de la façon dont les données sont injectées dans le modèle. Ainsi, le modèle est capable de faire face à une génération d'événements "en ligne" lorsque leur volume n'est pas trop gros. Des notifications peuvent également se produire dans le modèle, en direction des autres composants, et de propager les modifications de configuration correspondant aux commandes initiées par l'utilisateur.

B.3 Le Modèle Visuelle d'agrégation

Une autre préoccupation relative aux applications de la grille est qu'elles peuvent être composées d'un grand nombre de processus. Quelques analyses sont déjà possibles avec des applications à grand échelle [50], mais seulement au niveau d'un *cluster*. Plusieurs questions se posent dans des environnements de grille lors de l'analyse de ces applications. L'une est la grande quantité de données de trace, qui dépend de deux facteurs : le nombre d'entités de l'application, et la quantité de détails recueillis pour chaque entité. Une autre question est le passage à l'échelle des techniques de visualisation [26], qui doivent s'adapter à la quantité de données qui peuvent être affichées sur l'écran sans que l'utilisateur ne perde la capacité de comprendre ce qui est représenté.

Les techniques de visualisation des outils doivent également passer à l'échelle pour l'analyse des applications parallèles. Si on considère seulement la quantité des entités observées, les outils devraient être en mesure de représenter au moins quelques milliers de processus sur le même écran. Un certain nombre de détails doivent également être présents dans la représentation. Un exemple d'un manque de passage à l'échelle est la représentation espace-temps où la quantité de données à représenter est limitée par la résolution verticale des écrans d'ordinateurs.

Parmi les outils existants pour la visualisation, Vampir a une technique hiérarchique pour sa visualisation espace-temps qui augmente la quantité de processus qui peuvent être consultés en même temps. La technique fonctionne en agrégeant les comportements des processus en fonction de la représentation hiérarchique. Le problème de cette approche est que chaque niveau d'information est présenté différemment, ce qui rend difficile l'analyse de l'ensemble des points de vue. D'autres outils tels que Jumpshot et Pajé, grâce à une fenêtre de défilement, peuvent faire face à un grand nombre d'entités analysées. Cette technique a un impact négatif car le comportement de toutes les entités ne figure plus dans la même visualisation.

Notre approche utilise un intervalle de temps pour créer une structure hiérarchique qui représente le comportement de l'application pour la période sélectionnée. Nous utilisons ensuite la technique Treemap [42] pour créer une représentation visuelle de la structure. La technique proposée augmente le nombre d'entités qui peuvent être représentées en même temps, et permet une comparaison directe entre elles. En outre, nous présentons aussi un mécanisme d'agrégation qui peut être appliqué pour changer la visualisation quand il y a de nombreuses entités qui doivent être analysés dans le même écran. La combinaison de ces deux techniques permet d'atteindre une passage à l'échelle de l'analyse visuelle des applications parallèles.

B.3.1 L'algorithme de Tranche de Temps

L'objectif de l'algorithme de tranche de temps est de créer une structure hiérarchique qui reflète le comportement du programme pendant un temps donné. Pour ce faire, les sommets de la hiérarchie doivent être des valeurs qui sont calculées à partir de deux facteurs : la définition d'une tranche de temps et un résumé des événements pour chaque entité présente dans cette période. Différents réglages pour définir l'intervalle de temps sont possibles, allant des petites aux grandes plages.

Le résumé des événements se fait en tenant compte du temps spécifié et de l'information sur une entité, présente dans les données de trace. Le but est de trouver une valeur numérique qui

représente le comportement de chaque entité. Il existe différentes façons de définir cette valeur, comme la quantité de temps ou le nombre de changements d'état, ou de toute autre information qui peut être prise dans les traces. Le principe général de l'algorithme est d'ajouter séparément les valeurs de chaque type de données qui peuvent être trouvées pour une entité, et ensuite de réaliser une union de cette somme avec la tranche de temps utilisé.

B.3.2 Agrégation Visuelle

L'utilisation d'une représentation Treemap permet la passage à l'échelle de l'analyse. Cela signifie que si la taille de la plateforme affichée est augmentée, la représentation reste compréhensible du point de vue de l'utilisateur. Si ce passage à l'échelle se produit correctement dans la plupart des situations, la technique reste limitée par la taille de l'espace dédié à la représentation sur l'écran de l'ordinateur.

Le modèle d'agrégation essaie de surmonter cette limitation par le biais de la réorganisation de la hiérarchie à afficher. Il agit principalement par l'agrégation des valeurs des feuilles de l'arbre dans les noeuds intermédiaires. Avec cette approche, le rendu Treemap peut être interrompu à tout niveau, sans perdre l'information importante qui a été enregistrée dans les feuilles.

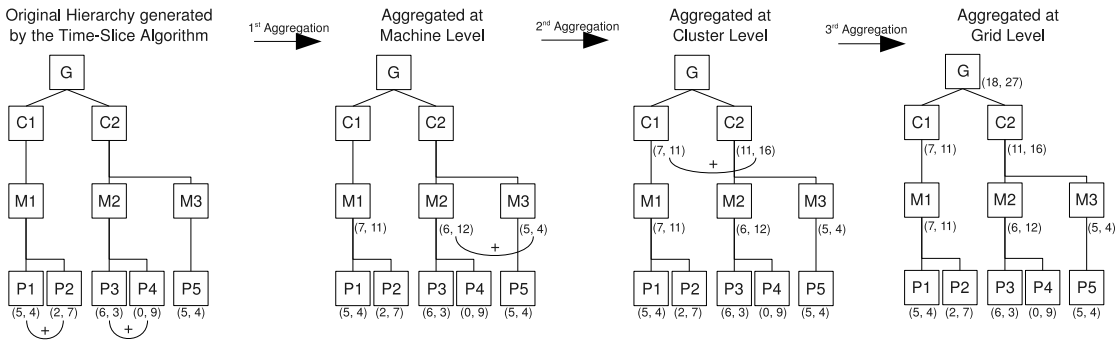


FIG. B.3 – Trois agrégations réalisées par le modèle.

La Figure B.3 montre trois changements dans la hiérarchie d'agrégation causés par le modèle. La hiérarchie originale est indiqué sur l'extrême gauche. Chaque information dans les feuilles peut représenter différentes métriques, telles que le nombre de fois où quelque chose se passe. Dans notre exemple, il existe trois niveaux intermédiaires : Processus (P), Machine (M) et Cluster (C). Le principal objectif de l'agrégation est de regrouper les valeurs de chaque processus et de les déplacer d'un niveau plus haut dans l'arbre. Par conséquent, après la première agrégation, les valeurs dans les vecteurs sont additionnées et stockées sur les noeuds machine. L'algorithme peut être appliqué de nouveau jusqu'à l'agrégation dans le noeud principal.

Outre l'opération d'addition (Figure B.3), le modèle d'agrégation peut être appliqué en utilisant d'autres fonctions telles que la teneur maximale, minimale, moyenne et médiane. L'application de ces fonctions dépend directement de la nature des informations agrégées et peut être utilisée pour mettre en évidence une caractéristique particulière.

Le bénéfice apporté par le modèle d'agrégation est évident quand il est combiné avec l'algorithme de la tranche de temps. Quand une application parallèle est composée de nombreux pro-

cessus, la technique de regroupement peut être utilisée pour améliorer l'analyse de l'affichage basé sur les treemaps.

B.4 L'implementation du Prototype Triva

Cette section décrit brièvement le prototype mis au point pour mettre en œuvre les modèles présentés dans les sections précédents. Cette description décrit les decisions d'implementation prises. Le prototype est appelé Triva.

L'un des principaux guides pour la réalisation de ce prototype est qu'il doit être construit à partir d'outils et de bibliothèques existants, en particulier afin de prévenir la ré-implementation d'outils déjà validés. La première décision est l'adoption de certaines parties de l'outil Pajé. Les principales raisons qui ont motivé cette adoption est la réutilisation de code et la performance de l'ensemble des composants Pajé. Les autres décisions prises sont notamment l'utilisation de formats de description des ressources faciles à reconnaître et l'adoption de la bibliothèque GraphViz.

La Figure B.4 montre l'organisation générale du prototype, composé de modules qui convertissent les données de trace pour des objets Pajé, puis élaborent les deux types de visualisation : la 3D et treemap. L'adoption des traces génériques a fait que la seule partie du prototype dépendante du format de la trace soit DIMVisual, représenté sur la gauche de la Figure. Les rectangles blancs sont des bibliothèques et des outils qui ont été réutilisés avec peu de changement ; rectangles gris ont été développés pour composer le prototype.

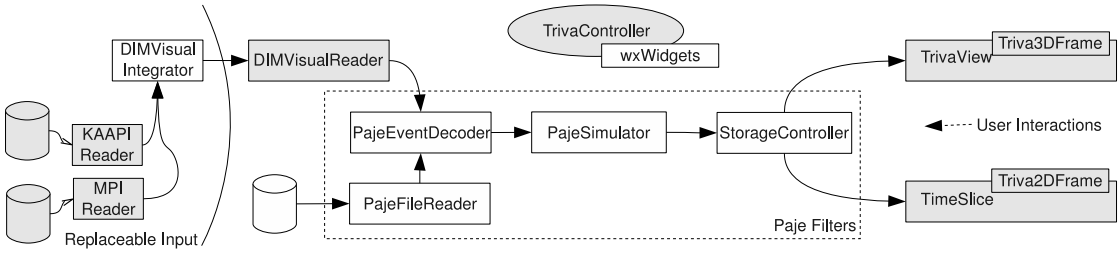


FIG. B.4 – L'architecture du prototype Triva.

Le composant TrivaController, écrit en langage C++, est en charge de la mise en route de tous les composants en les reliant selon l'architecture de la Figure B.4. Elle fournit également à l'utilisateur une interface graphique, créée en utilisant la bibliothèque wxWidgets, sous la forme d'une fenêtre, avec des options de configuration et des mécanismes d'interaction. Les visualisations 3D et treemap sont aussi mises en route par cette composante.

Les filtres, représentés par des rectangles en pointillés dans la Figure B.4, sont les mêmes filtres que ceux utilisés dans Pajé. Leur implementation prend en compte plusieurs questions liées au passage à l'échelle et au temps de réponse des commandes de l'interface utilisateur. Le premier de ces filtres, PajeEventDecoder, traite l'entrée générée par DIMVisualReader et la prépare pour le prochain module. Le PajeSimulator transforme les événements en objets visuels. Cette transformation a comme but la creation d'une structure hiérarchique de trace, en utilisant les types de base Pajé. Cette structure, qui représente la même information que celle qui se trouve dans les fichiers d'entrée, est optimisée pour la visualisation, et enregistrée dans le StorageController.

Dans la partie droite de la Figure B.4, les interactions entre les modules opèrent dans les deux

directions. Les interactions de la droite vers la gauche sont les demandes de nouvelles données. Elles sont initiées par l'utilisateur par des commandes ou par la modification de paramètres. Les interactions de gauche à droite sont des réponses à des demandes.

B.4.1 TrivaView

Le modèle de visualisation 3D, présenté dans la section B.2, est mis en oeuvre dans le prototype Triva par la composante TrivaView. La Figure B.5 montre l'organisation globale de cette composante. Le module implémente la partie Extractor du modèle. Il obtient du flot des objets Pajé les conteneurs et les liens à envoyer au EntityMatcher, et envoie aussi le flot au composant DrawManager. La partie du modèle 3D appelé EntityMatcher est mise en œuvre dans les trois composantes du prototype : TrivaApplicationGraph, TrivaResourcesGraph et TrivaTreemapSquarified. Ils reçoivent en plus du flot d'objects Pajé, le fichier de description des ressources. La visualisation du modèle 3D, representée dans la droite de la Figure B.5, est mise en œuvre avec 4 composantes : le Triva3DFrame, pour le maintien de la scène 3D, et de trois mainteneurs qui peuvent changer les aspects visuels de la scéne, le DrawManager, le AmbientManager et le CameraManager.

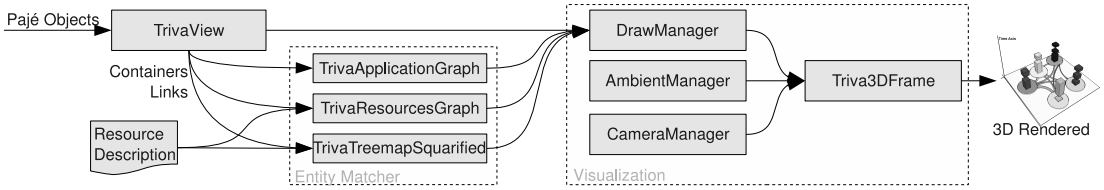


FIG. B.5 – Structure d'implementation du TrivaView.

B.4.2 TimeSliceView

Le modèle d'agrégation et l'algorithme de tranche de temps ont été mis en œuvre dans le composant TimeSliceView, comme le montre la Figure B.6. Une autre composante importante de cette partie est le Triva2DFrame, dont la responsabilité est de dessiner le treemap dans la fenêtre de visualisation du prototype.

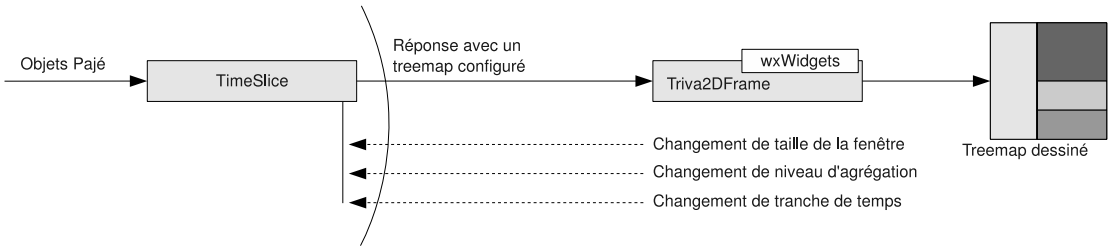


FIG. B.6 – Structure d'implementation du TimeSliceView.

La Figure B.6 détaille également les interactions et les notifications qui se produisent pendant l'exécution de la composante. L'arrivée d'objets du simulateur Pajé (voir Figure B.4 pour plus de détails) est représentée sur la gauche de la Figure. Les interactions avec l'utilisateur peuvent provoquer des trois différents types de notifications qui partent de la composante Triva2DFrame vers la composante TimeSlice : changer la taille de la fenêtre, changer de niveau d'agrégation ou modifier la tranche de temps. Toutes ces notifications déclenchent la même chaîne d'événements dans le composant : la création d'une hiérarchie de comportement, l'application possible des opérateurs d'agrégation et le calcul des nouveaux treemap. Le treemap résultant est envoyé comme une réponse et est dessiné dans la fenêtre par le composant Triva2DFrame.

B.5 Résultats Obtenus et Évaluation

L'objectif principal de cette partie du résumé est de présenter les deux types de visualisations générées par le prototype, une en trois dimensions et l'autre sous forme de treemap. Dans le même temps, une analyse de ces resultats est faite compte tenu des traces d'exécution utilisées comme entrée pour le prototype.

B.5.1 Trois Dimensions

Le principal objectif du modèle 3D est de réaliser la combinaison en trois dimensions des composants de l'application avec la topologie d'interconnexion des ressources. Dans un premier temps, nous avons retenu un scénario comprenant 60 processus, divisés en 2 sites différents de Grid'5000. Le site *nancy* contribue à la l'exécution avec 30 machines du cluster *grelon*, tandis que le site *rennes* a une allocation de 25 machines du cluster *paramount* et 5 machines du cluster *paraquad*. Nous considérons ici une topologie du réseau dans laquelle chaque site contient un routeur lui-même et tous les *clusters* sont connectés au routeur de leur site. Les routeurs de différents sites sont interconnectés via un backbone. Ainsi, quand un message est envoyé à un *cluster* d'un site à partir d'un *cluster* d'un autre site, il doit passer par l'intermédiaire de deux routeurs.

La Figure B.7 montre deux captures d'écran du prototype Triva générées lors de l'affichage du fichier de trace de ce scénario. Le texte et les lignes en pointillés ont été ajoutés manuellement pour accroître la compréhension de l'exemple. L'image **A** montre le temps total d'exécution avec une petite échelle de temps, de sorte que tous les objets soient dans la base de la visualisation. La ligne pointillée montre la séparation entre les sites *rennes*, avec deux *clusters*, et *nancy*, avec un seul *cluster*. Nous pouvons voir à cette échelle de temps, un grand nombre de vols de travail entre les groupes *grelon* et *paraquad*, probablement dû au nombre de processus qui leur sont attribués. L'analyse de ces interactions en conjonction avec la topologie du réseau permet à l'utilisateur de voir que toutes les demandes de travail de ces *clusters* doivent passer à travers les deux routeurs de l'interconnexion.

Le prototype permet également de changer de façon dynamique l'échelle de temps, en utilisant la souris. L'image **B** dans la Figure B.7 indique le temps d'exécution total pour les traces de ce scénario, mais avec une plus grande échelle de temps. Grâce à cette image, il est possible d'observer des différences dans le comportement de vol de travail à différentes périodes de temps de l'exécution. Il est ainsi possible d'apercevoir qu'au début, il y a beaucoup moins de vols qu'à la fin. La raison de cela est qu'à la fin d'une application KAAPI, les tâches deviennent plus rares. Ce comportement est normal, vu que le vol de travail des tâches implémenté dans la version actuelle de la bibliothèque KAAPI est aléatoire.

Un deuxième scénario est une application KAAPI composée de 200 processus sur 200 machines. La répartition des machines est divisée en deux sites : *rennes* et *nancy*. Le nombre de machines affectées à chacun d'eux est le même, bien que la répartition interne de chacun diffère au niveau du nombre de machines par cluster. L'image **A** de la Figure B.8 indique le nombre de machines affectées à chaque *cluster* ainsi que la topologie du réseau qui relie les deux sites. La ligne pointillée est utilisé pour separer les sites distincts. Nous considérons pour ce scénario

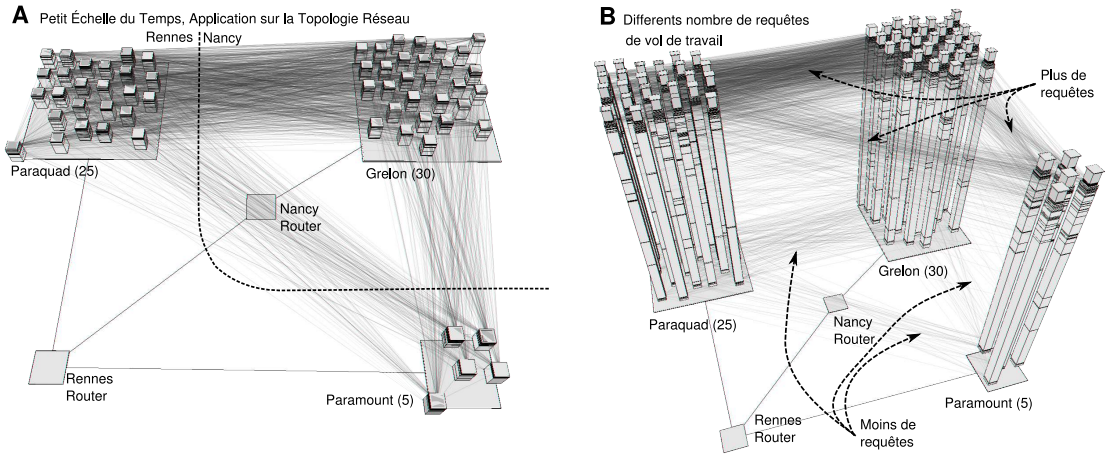


FIG. B.7 – Deux captures d'écran du prototype Triva pendant la visualisation d'une application composé de 60 processus, à différentes échelles de temps.

l'existence d'informations complémentaires concernant l'interconnexion entre les routeurs et les trois *clusters*. La bande passante disponible entre les *clusters paravent* et *grillon*, à travers les deux routeurs, est de 100 mégabits. Le lien entre le *cluster grelon* et son routeur est de 1 mégabit, comme indiqué dans l'image **A** de la Figure.

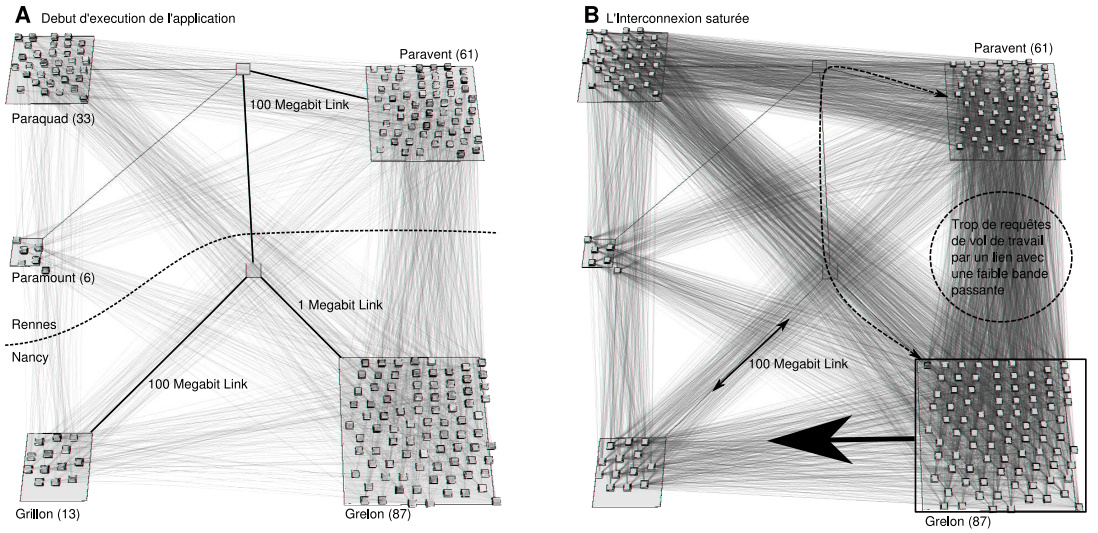


FIG. B.8 – Deux vues d'un exemple avec plus d'informations sur la topologie du réseau, telles que les limitations imposées par la bande passante.

Dans ce scénario, il y a 87 processus en cours dans le cluster *grelon*, et 61 dans *paravent*. Considérons seulement les vols du travail entre ces deux *clusters*, indiqués dans le cercle en pointillés de l'image sur la droite de la Figure B.8 : la flèche en pointillés dans la même image

indique que ces demandes doivent passer par le lien de 1 mégabit. La visualisation permet de déduire qu'un nombre plus restreint de processus devraient être placés dans un *cluster* avec une bande passante limitée. Si, par exemple, les processus du cluster *grelon* pouvaient être exécutés dans le cluster *grillon*, l'application pourrait atteindre une meilleure performance.

À travers l'exemple de ce deuxième scénario, nous pouvons noter l'importance d'analyser la performance d'une application accompagnée d'une représentation topologique du réseau. Si ce type de visualisation, illustré dans l'image **B** dans la Figure B.8, n'est pas présent, l'analyste peut obtenir des conclusions incomplétes sur les performances de l'application.

B.5.2 Agrégation

Un des principaux avantages du modèle d'agrégation de données de cette thèse est la facilité qu'il apporte pour l'analyse d'un grand nombre d'entités dans le même écran. Pour évaluer la façon dont la visualisation passe à l'échelle, une trace de synthèse composée de 100 milliers de processus a été utilisé, chacun avec deux états différents. L'analyse qui suit montre l'emploi de la technique de la tranche de temps et de l'algorithme d'agrégation.

La Figure B.9 montre l'analyse de cette trace, qui comprend une hiérarchie à quatre niveaux : Site, Cluster, Machine et Processor. La hiérarchie contient 10 sites, chacun avec 10 clusters, chaque cluster avec 100 machines et chaque machine avec 100 processus. Chaque processeur peut être dans l'un des deux états possibles, représentée dans la Figure par les différentes tonalités de gris.

L'analyse à grande échelle en utilisant le prototype commence avec le treemap **A**, situé en haut à gauche de la Figure B.9, avec le niveau Processor. Dans ce treemap, il y a 200 mille rectangles : 100 mille fois le nombre des états possibles. Nous pouvons observer que certaines régions du treemap sont plus sombres que d'autres, permettant une sorte de conclusion concernant la repartition des états. Toutefois, une conclusion précise est difficile à atteindre avec cette representation. La raison principale est que le treemap **A** comporte des rectangles qui sont très petits, de sorte qu'il est difficile de noter des différences de taille entre deux états d'un seul processeur. L'exemple est montré pour illustrer la limitation de la visualisation treemap traditionnelle.

Le rectangle blanc de la treemap **A** dans la Figure B.9 représente l'espace dédié à une machine. Bien qu'il soit difficile de le constater, il y a 200 rectangles dans cette petite région qui représente l'état des 100 processeurs de cette machine. Comme il est difficile de comprendre la structure de l'ensemble de ces 100 processus, l'utilisateur peut interagir avec le prototype et visualiser la valeur agrégée au niveau de la machine, comme montré dans le treemap **B** de la Figure. Elle indique, pour chaque machine, les deux états possibles. Dans cette representation, il est possible d'examiner les différences entre les machines : certaines passent beaucoup plus de temps dans un état que dans un autre. La zone en évidence sur le côté gauche du treemap **B**, présentée par l'intermédiaire d'un zoom, est la zone du rectangle blanc du treemap **A**.

Les agrégations suivantes permettent à l'utilisateur de visualiser les traces au niveau Cluster, comme indiqué dans le treemap **C** de la même Figure, et au niveau Site dans le treemap **D**. Le treemap C montre 100 clusters (10 par site). Dans sa partie gauche, le treemap contient un rectangle noir qui montre les 10 clusters dans la région dédiée à un site. La flèche dans ce rectangle désigne les valeurs agrégés pour ce site, dans le treemap D. L'agrégation maximale possible, le treemap **E**, permet d'avoir une vue des informations d'état disponibles globalement,

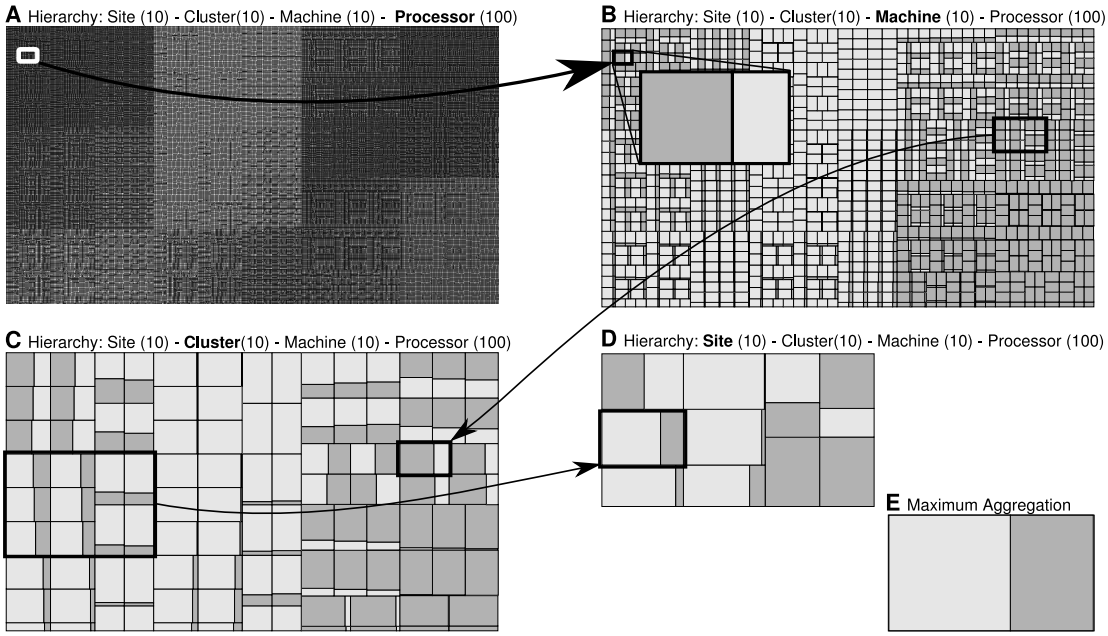


FIG. B.9 – Visualisation treemap normale (A) and quatre agrégés (B – E) de deux états pour 100 mille processeurs (trace synthétique).

en indiquant que l'état représenté par le ton plus claire apparaît plus souvent que l'autre dans la tranche de temps choisie pour cet exemple.

Un second scénario pour la visualisation treemap est une application de 188 processus, réalisée sur 188 machines, réparties dans cinq sites du Grid'5000, dont le site de Porto Alegre. Il y a 13 machines affectées dans le cluster *xiru*, à *portoalegre* ; 2 de *bordereau*, 17 de *bordemer*, et 6 de *bordeplage*, à *bordeaux* ; 45 de *pastel*, 5 de *violette*, à *toulouse* ; 14 de *paramount*, 36 de *paraquad*, à *rennes* ; et finalement 50 de *grelon* à *nancy*. La Figure B.10 montre deux treemaps calculés avec les traces générées dans ce scénario.

Le treemap **A** montre les états Run et RSteal pour les 188 processus. Presque tous les processus exhibent le même comportement, avec plus de temps passé dans l'état Run (zones avec un ton de gris clair) par rapport à l'état RSteal (gris foncé). La seule exception apparaît dans le K-processus au sein du site de *portoalegre*, manuellement mis en evidence avec le cercle en pointillés. Nous notons que ces processus sont restés plus de temps à voler les tâches que les processus d'autres sites. Le treemap **B**, à droite, montre la même tranche de temps et les mêmes processus, mais seulement pour l'état RSteal. Ici, la différence de temps passé à voler les tâche devient encore plus évidente. Nous pensons que la principale raison de ce comportement vient de l'interconnexion entre les sites. Le site de *portoalegre* se trouve au Brésil, et son lien avec Grid'5000 n'est fait qu'à travers d'un réseau privé virtuel (VPN) qui est maintenue grâce à Internet. La latence de cette connexion, par rapport à la latence globale entre les sites Grid'5000 situés en France, est significative. Le vol des tâches traditionnellement mis en œuvre dans KAAPI differencie pas les cibles d'un vol. Ce choix, dans un environnement d'interconnexion hétérogène,

peut conduire à passer plus de temps à voler, comme indiqué par le treemap calculé par notre algorithme de tranche de temps.

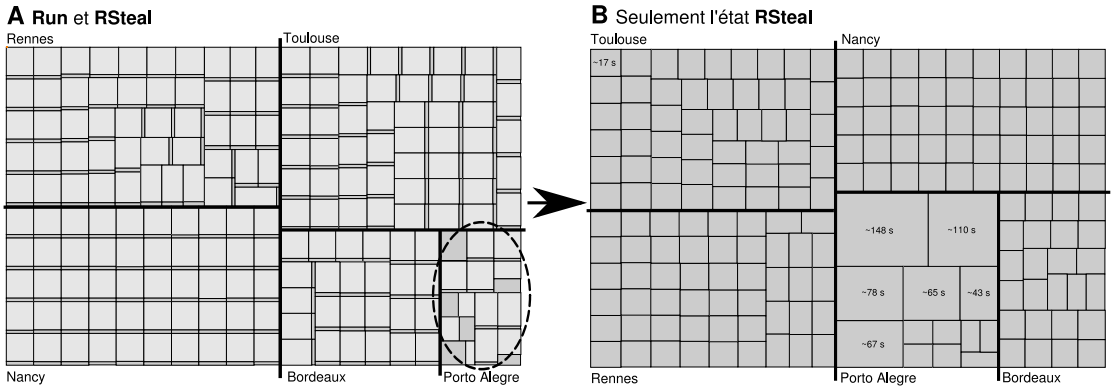


FIG. B.10 – Scénario KAAPI avec une application composée de 188 processus.

Globalement, l'algorithme de tranche de temps combiné avec le modèle d'agrégation de cette thèse permetent d'identifier facilement les problèmes de performance lorsqu'on compare le comportement relatif des processus dans une application parallèle. Le modèle d'agrégation présente des avantages pour les situations à grande échelle, peu importe le nombre de processus impliqués dans l'analyse. La seule mesure nécessaire pour permettre l'application de nos propositions est la définition d'une hiérarchie avec au moins quelques niveaux. Le hiérarchie utilisée dans le scénario KAAPI dispose de 5 niveaux, ce qui permet d'obtenir de bons résultats dans la visualisation.

B.6 Conclusion

Les visualisations classiques pour l'analyse des applications parallèles sont conçues pour traiter des données à petite échelle et équilibrées. Le besoin de techniques de visualisation pour l'analyse à grande échelle, telles que au sein de grilles de calcul, motive ce travail. Deux problèmes dans l'analyse des applications parallèles par le biais de la visualisation sont soulevés dans cette thèse.

Le premier est l'impact de l'interconnexion des réseaux dans l'exécution des applications parallèles. Cet impact devrait être pris en compte dans l'analyse pour mieux comprendre et améliorer les performances de l'application. Les techniques traditionnelles de visualisation, tels que les graphiques espace-temps par exemple, sont largement utilisés pour l'analyse des applications. Toutefois, ces techniques ne peuvent pas montrer, dans le même affichage, la topologie du réseau et le suivi des données d'exécution de l'application. Cela peut conduire à des conclusions erronées dans la détection des problèmes de performance des applications. Le deuxième problème est la passage à l'échelle des techniques de visualisation. Généralement, le nombre d'entités de suivi que l'on peut voir sur le même écran est limité à la résolution verticale de l'écran d'un ordinateur. Les représentations espace-temps en 2D sont un exemple clair de ce problème, elles sont mal adaptées à l'analyse des applications de grille composées d'un grand nombre de processus.

L'idée principale de cette thèse est l'exploitation des techniques de visualisation d'information qui peuvent être utilisées pour analyser le comportement des applications parallèles. Notre première approche montre le réseau d'interconnexion, ainsi que des données de l'application en utilisant une vue en trois dimensions. La base de ce point de vue est utilisée pour detailler l'interconnexion entre les ressources, et le troisième axe pour montrer l'évolution de l'application dans le temps. Cette visualisation est complétée par la représentation des communication, qui donne la possibilité au développeur de les comparer avec la topologie du réseau.

La deuxième approche est le modèle visuel d'agrégation, où les problèmes de passage à l'échelle sont surmontés par la combinaison de la technique du treemap et de l'algorithme de tranche de temps. Cet algorithme prend en compte des tranches de temps pour générer des valeurs et de les injecter dans une organisation hiérarchique de l'application. Cette structure est alors représentée par la technique du treemap. Le passage à l'échelle est réalisé par le modèle d'agrégation, où les niveaux de la hiérarchie sont utilisées pour créer des données intermédiaires qui peuvent être utilisés pour une représentation treemap avec plus d'informations.

Les deux approches ont été implémentées dans un prototype appelé Triva, développé en utilisant un gestionnaire de scènes 3D appelé Ogre et une implementation de l'algorithme Treemap. Le prototype dispose de mécanismes pour la lecture des traces fournis par la bibliothèque DIMVisual, capable d'intégrer les données provenant de différentes sources et formats. Des traces synthétiques et réelles d'applications KAAPI et MPI ont été utilisées pour valider l'approche et l'implementation. Le traces KAAPI ont été recueillies sur la plate-forme Grid'5000. Bien que l'évaluation du prototype est liée à l'analyse d'applications KAAPI et MPI, le format d'entrée Pajé permet d'étendre les avantages de l'outil à d'autres domaines de recherche, pour visualiser d'autres types de ressources dans les bibliothèques de communication.

Les résultats sont prometteurs. La visualisation en trois dimensions permet de mieux com-

prendre les communications en conjonction avec la topologie du réseau. En ayant recours à une simplification de la topologie de Grid'5000, nous avons pu montrer que dans les différentes tranches de temps, le vol de travail dans KAAPI pourrait bénéficier davantage de la localité. En effet, l'implementation actuelle de KAAPI ne prend pas en compte le réseau pour faire les requêtes de vol de travail. En outre, les résultats obtenus avec le modèle d'agrégation ont permis la visualisation des états de 100 milliers de processeurs, générés de manière synthétique. Les treemaps définis par l'algorithme de la tranche de temps ont également été déterminés en utilisant des traces KAAPI et MPI. Nous avons été en mesure d'identifier dans les traces KAAPI des caractéristiques variées, telles que le comportement de différents mécanismes de vol effectuées par des processus distincts, l'efficacité de l'équilibrage de la charge pour l'ensemble du temps d'exécution de applications, et l'analyse d'une application KAAPI à grande échelle composée de près de 3 mille processus.

En résumé, les principaux objectifs atteints dans cette thèse sont la proposition d'une approche tridimensionnelle, le modèle visuel d'agrégation combiné avec la tranche de temps et le prototype Triva. En outre, il comprend l'interaction entre Triva et la bibliothèque KAAPI, permettant une analyse des activités de vol de travail de cette bibliothèque.

Comme perspectives, il est prévu l'extension de la visualisation 3D pour la représentation de l'information produite par le modèle d'agrégation, la création des graphes d'application réduits avec la technique de la tranche de temps et d'agrégation, l'étude d'autres fonctions d'agrégation et l'utilisation d'autres données pour l'algorithme de la tranche de temps. Nous pensons que la plus importante contribution de ce travail est l'étude des techniques du domaine de la visualisation appliquées à l'analyse des applications parallèles.

Bibliography

[1] David P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID'04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[2] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[3] Apple. *Property List Programming Guide for Cocoa: Introduction to Property Lists*. Cocoa Developer Connection, November 2008. `http://developer.apple.com/documentation/Cocoa/Conceptual/PropertyLists/PropertyLists.pdf`. Last accessed October 7, 2009.

[4] D.C. Arnold, D.H. Ahn, B.R. de Supinski, G.L. Lee, B.P. Miller, and M. Schulz. Stack trace analysis for large scale debugging. In *The Proceedings of the IEEE International Parallel and Distributed Processing Symposium. IPDPS 2007*, March 2007.

[5] Dorian C. Arnold, Dong H. Ahn, Bronis R. de Supinski, Gregory L. Lee, Barton P. Miller, and Martin Schulz. Stack trace analysis for large scale debugging. In *IPDPS 2007: Proceedings of the 2007 International Parallel and Distributed Processing Symposium*, page 64, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[6] R. A. Aydt. Sddf: The pablo self-describing data format. Technical report, Department of Computer Science, University of Illinois, September 1993.

[7] David F. Bacon, Perry Cheng, Daniel Frampton, and David Grove. Tuningfork: Visualization, analysis and debugging of complex real-time systems. Technical Report IBM Research Report RC24162, IBM, 2007.

[8] DH Bailey, E. Barszcz, JT Barton, DS Browning, RL Carter, L. Dagum, RA Fatoohi, PO Frederickson, TA Lasinski, RS Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63, 1991.

[9] Michael Balzer, Oliver Deussen, and Claus Lewerentz. Voronoi treemaps for the visualization of software metrics. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 165–172, New York, NY, USA, 2005. ACM Press.

[10] R. Bell, A.D. Malony, and S. Shende. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. *Lecture Notes in Computer Science*, pages 17–26, 2003.

[11] X. Besseron, S. Bouguerra, T. Gautier, E. Saule, and D. Trystram. *Fault tolerance and availability awarness in computational grids*, chapter 5. Numerical Analysis and Scientific Computing. Chapman and Hall/CRC Press, to appear 2009. ISBN: 978-1439803677.

[12] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lantéri, J. Leduc, N. Melab, G. Mornet andR. Namyst, P. Primet, B. Quetier, O. Richard, E-G. Talbi, and I. Touche. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

[13] M. Bruls, K. Huizing, and J. van Wijk. Squarified treemaps. In *Proceedings of Joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42. IEEE Press, 2000.

[14] H. Brunst, H.C. Hoppe, W.E. Nagel, and M. Winkler. Performance optimization for large scale computing: The scalable vampir approach. In *Proceedings of the International Conference on Computational Science-Part II*, pages 751–760. Springer-Verlag London, UK, 2001.

[15] M.S.T. Carpendale, D.J. Cowperthwaite, and F.D. Fracchia. Extending distortion viewing from 2d to 3d. *IEEE Computer Graphics and Applications*, 17(4):42–51, 1997.

[16] Charlie Catlett. The philosophy of teragrid: Building an open, extensible, distributed terascale facility. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 8, Washington, DC, USA, 2002. IEEE Computer Society.

[17] A. Chan, W. Gropp, and E. Lusk. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, 16(2):155–165, 2008.

[18] Christian Clémençon, Akiyoshi Endo, Josef Fritscher, Andreas Müller, and Brian J. N. Wylie. Annai scalable run-time support for interactive debugging and performance analysis of large-scale parallel programs. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par, Vol. I*, volume 1123 of *Lecture Notes in Computer Science*, pages 64–69. Springer, 1996.

[19] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: Concepts and Design*. Addison-Wesley Reading, Mass, 4th edition, 2005.

[20] Gabriela Jacques da Silva, Lucas Mello Schnorr, and Benhur Stein. Jrastro: A trace agent for debugging multithreaded and distributed java programs. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, pages 46–54. Los Alamitos: IEEE Computer Society, 2003.

[21] J. Chassin de Kergommeaux and B. de Oliveira Stein. Flexible performance visualization of parallel and distributed applications. *Future Generation Computer Systems*, 19(5):735–747, 2003.

[22] J.C. de Kergommeaux, B. Stein, and PE Bernard. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26(10):1253–1274, 2000.

[23] B. de Oliveira Stein, J.C. de Kergommeaux, and G. Mounié. Pajé trace file format. Technical report, ID-IMAG, Grenoble, France, 2002. `http://www-id.imag.fr/Logiciels/paje/publications`.

[24] Benhur de Oliveira Stein. *Visualisation interactive et extensible de programmes parallèles à base de processus légers*. PhD thesis, Université Joseph Fourier, 1999.

[25] Wim De Pauw, Henrique Andrade, and Lisa Amini. Streamsight: a visualization tool for large-scale streaming applications. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 125–134, New York, NY, USA, 2008. ACM.

[26] S.G. Eick and A.F. Karr. Visual scalability. *Journal of Computational and Graphical Statistics*, 11(1):22–43, 2002.

[27] John Ellson, Emden Gansner, Lefteris Koutsofios, North Stephen C., and Gordon Woodhull. Graphviz-open source graph drawing tools. *Lecture Notes in Computer Science*, 2265:594–597, 2002.

[28] Jean-Daniel Fekete and Catherine Plaisant. Interactive information visualization of a million items. In *INFOVIS '02: Proceedings of the IEEE Symposium on Information Visualization (InfoVis'02)*, page 117, Washington, DC, USA, 2002. IEEE Computer Society.

[29] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[30] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 1-558-60933-4.

[31] Henrique Freitas, Marco Alves, Lucas Mello Schnorr, and Philippe Olivier Alexandre Navaux. Performance evaluation of noc architectures for parallel workloads. In *The 3rd ACM/IEEE International Symposium on Networks-on-Chip, NOCS*. IEEE Computer Society, 2009.

[32] F. Galilée, J.L. Roch, G.G.H. Cavalheiro, and M. Doreille. Athapascan-1: On-line building data flow graph in a parallel language. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society Washington, DC, USA, 1998.

[33] Emden R. Gansner. Drawing graphs with graphviz. Technical report, GraphViz WebSite, April 2009. `http://www.graphviz.org/pdf/libguide.pdf`. Last accessed June, 17, 2009.

[34] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software Practice Experience*, 30(11):1203–1233, 2000.

[35] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the international workshop on Parallel symbolic computation*, pages 15–23, New York, NY, USA, 2007. ACM.

[36] GA Geist, MT Heath, BW Peyton, and PH Worley. A user's guide to picl a portable instrumented communication library. Technical report, ORNL/TM-11616, Oak Ridge National Lab., TN (USA), 1990.

[37] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-57104-8.

[38] MT Heath and JA Etheridge. Visualizing the performance of parallel programs. *IEEE software*, 8(5):29–39, 1991.

[39] S. Heisig. Treemaps for workload visualization. *Computer Graphics and Applications, IEEE*, 23(2):60–67, 2003.

[40] Jeffrey K. Hollingsworth. *Finding Bottlenecks in Large-scale Parallel Programs*. PhD thesis, University of Wisconsin – Madison, August 1994.

[41] K.A. Huck and A.D. Malony. Perfexplorer: A performance data mining framework for large-scale parallel computing. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Washington, DC, USA, 2005.

[42] B. Johnson and B. Shneiderman. *Tree-Maps: a space-filling approach to the visualization of hierarchical information structures*. IEEE Computer Society Press Los Alamitos, CA, USA, 1991.

[43] Gregory Junker. *Pro OGRE 3D Programming (Pro)*. Apress, Berkely, CA, USA, 2006.

[44] Laxmikant V. Kalé, Gengbin Zheng, Chee Wai Lee, and Sameer Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. *Future Generation Comp. Syst.*, 22(3):347–358, 2006.

[45] L.V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108. ACM New York, NY, USA, 1993.

[46] I.H. Kazi, D.P. Jose, B. Ben-Hamida, C.J. Hescott, C. Kwok, J.A. Konstan, D.J. Lilja, and P.C. Yew. Javiz: A client/server java profiling tool. *IBM Systems Journal*, 39(1):96–117, 2000.

[47] Jacques Chassin de Kergommeaux and Benhur de Oliveira Stein. Pajé: An extensible environment for visualizing multi-threaded programs executions. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 133–140, London, UK, 2000. Springer-Verlag.

[48] Derrick Kondo, Michela Taufer, Charles L. Brooks III, Henri Casanova, and Andrew A. Chien. Characterizing and evaluating desktop grids: An empirical study. *International Parallel and Distributed Processing Symposium*, 1:26b, 2004.

[49] S. Lacour, C. Pérez, and T. Priol. A network topology description model for grid application deployment. In *Proceedings of the 5th IEEE/ACM International Conference on Grid Computing*, pages 61–68. IEEE Computer Society Washington, DC, USA, 2004.

[50] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Matthew Legendre, Barton P. Miller, Martin Schulz, and Ben Liblit. Lessons learned at 208k: towards debugging millions of cores. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press.

[51] T. Lehr, Z. Segall, D.F. Vrsalovic, E. Caplan, A.L. Chung, and C.E. Fineman. Visualizing performance debugging. *Computer*, 22(10):38–51, Oct 1989.

[52] E. Maillet and C. Tron. On efficiently implementing global time for performance evaluation on multiprocessor systems. *Journal of Parallel and Distributed Computing*, 28(1):84–93, 1995.

[53] AD Malony, DH Hammerslag, and DJ Jablonowski. Traceview: a trace visualization tool. *Software, IEEE*, 8(5):19–28, 1991.

[54] S. Mansmann, F.; Vinnik. Interactive exploration of data traffic with hierarchical network maps. *IEEE Transactions on Visualization and Computer Graphics*, 12(6):1440–1449, Nov-Dec 2006.

[55] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[56] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. S. Lim, and T. Torzewski. Ips-2: The second generation of a parallel program measurement system. *IEEE Trans. Parallel Distrib. Syst.*, 1(2):206–217, 1990.

[57] Barton P. Miller, Mark D. Callaghan, Joanthan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[58] Barton P. Miller and Cui-Qing Yang. Ips: An interactive and automatic performance measurement tool for parallel and distributed programs. In *ICDCS*, pages 482–489, 1987.

[59] B. Mohr and F. Wolf. Kojak-a tool set for automatic performance analysis of parallel programs. *Lecture notes in computer science*, pages 1301–1304, 2003.

[60] W.E. Nagel, A. Arnold, M. Weber, H.C. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources. *Supercomputer*, 12(1):69–80, 1996.

[61] G. Nagy and S. Seth. Hierarchical representation of optically scanned documents. In *Proceedings of International Conference on Pattern Recognition*, volume 1, pages 347–349, 1984.

[62] F.G. Ottogalli, C. Labbé, V. Olive, B. de Oliveira Stein, J.C. de Kergommeaux, and J.M. Vincent. Visualisation of distributed applications for performance debugging. In *Proceedings of the International Conference on Computational Science-Part II*, pages 831–840. Springer-Verlag London, UK, 2001.

[63] V. Pillet, J. Labarta, T. Cortes, and S. Girona. Paraver: A tool to visualise and analyze parallel code. In *Proceedings of Transputer and occam Developments, WOTUG-18.*, volume 44 of *Transputer and Occam Engineering*, pages 17–31, Amsterdam, 1995. [S.l.]: IOS Press.

[64] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.*, pages 104–113. IEEE Computer Society, 1993.

[65] D.A. Reed, R.A. Aydt, T.M. Madhyastha, R.J. Noe, K.A. Shields, and B.W. Schwartz. An overview of the pablo performance analysis environment. Technical report, Department of Computer Science, University of Illinois, 1992.

[66] DA Reed, PC Roth, RA Aydt, KA Shields, LF Tavera, RJ Noe, and BW Schwartz. Scalable performance analysis: the pablo performance analysisenvironment. In *Scalable Parallel Libraries Conference, 1993., Proceedings of the*, pages 104–113, 1993.

[67] R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed. Autopilot: Adaptive control of distributed applications. In *HPDC '98: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, page 172, Washington, DC, USA, 1998. IEEE Computer Society.

[68] Sratha Saengsuwarn and Vivek Pai. Covisualize - visualization of planetlab project, February 2009. http://codeen.cs.princeton.edu/covisualize/. Last accessed July, 28, 2009.

[69] Manojit Sarkar and Marc H. Brown. Graphical fisheye views. *Commun. ACM*, 37(12):73–83, 1994.

[70] Lucas Mello Schnorr, Philippe O. A. Navaux, and Benhur de Oliveira Stein. Dimvisual: Data integration model for visualization of parallel programs behavior. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 473–480, Washington, DC, USA, 2006. IEEE Computer Society.

[71] Z. Segall and L. Rudolph. Pie: A programming and instrumentation environment for parallel processing. *IEEE Software*, 2(6):22–37, 1985.

[72] Eric Shaffer, Daniel A. Reed, Shannon Whitmore, and Benjamin Schaeffer. Virtue: Performance visualization of parallel and distributed applications. *Computer*, 32(12):44–51, 1999.

[73] B. Shneiderman and M. Wattenberg. Ordered treemap layouts. In *Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*. IEEE Computer Society Washington, DC, USA, 2001.

[74] Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.*, 11(1):92–99, 1992.

[75] J. Smart, K. Hock, and S. Csomor. *Cross-Platform GUI Programming with wxWidgets (Bruce Perens Open Source)*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.

[76] Benhur Stein, Jacques Chassin de Kergommeaux, and Pierre-Eric Bernard. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26:1253–1274, 2000.

[77] Martin Wattenberg. Visualizing the stock market. In *CHI '99: CHI '99 extended abstracts on Human factors in computing systems*, pages 188–189, New York, NY, USA, 1999. ACM Press.

[78] Jarke J. Van Wijk and Huub van de Wetering. Cushion treemaps: Visualization of hierarchical information. In *INFOVIS '99: Proceedings of the 1999 IEEE Symposium on Information Visualization*, page 73, Washington, DC, USA, 1999. IEEE Computer Society.

[79] James M. Wilson. Gantt charts: A centenary appreciation. *European Journal of Operational Research*, 149(2):430–437, September 2003.

[80] F. Wolf and B. Mohr. Automatic performance analysis of hybrid mpi/openmp applications. *Journal of Systems Architecture*, 49(10-11):421–439, 2003.

[81] J. C. Yan. Performance tuning with AIMS — an Automated Instrumentation and Monitoring System for multicomputers. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, pages 625–633, 1994.

[82] Cui-Qing Yang and Barton P. Miller. Performance measurement for parallel and distributed programs: A structured and automatic approach. *IEEE Trans. Software Eng.*, 15(12):1615–1629, 1989.

[83] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with jumpshot. *Int. J. High Perform. Comput. Appl.*, 13(3):277–288, 1999.

Title: Some Visualization Models applied to the Analysis of Parallel Applications.
Abstract: This thesis tries to overcome the issues encountered on traditional visualization techniques for parallel applications. The main idea behind our efforts is to explore techniques from the information visualization research area and to apply them in the context of parallel applications analysis. Based on this main idea, the thesis proposes two visualization models: the three-dimensional and the visual aggregation model. The former might be used to analyze parallel applications taking into account the network topology of the resources. The visualization itself is composed of three dimensions, where two of them are used to render the topology and the third is used to represent time. The later model can be used to analyze parallel applications composed of several thousands of processes. It uses hierarchical organization of monitoring data and an information visualization technique called Treemap to represent that hierarchy.
Keywords: Parallel applications, performance analysis, visualization, 3D visualization, treemap, scalability, grid.

Título: Alguns Modelos de Visualização aplicados para a Análise de Aplicações Paralelas.
Resumo: Esta tese tenta resolver os problemas encontrados em técnicas de visualização tradicionais para a análise de aplicações paralelas. A idéia principal consiste em explorar técnicas da área de visualização da informação e aplicá-las no contexto de análise de programas paralelos. Levando em conta isto, esta tese propõe dois modelos de visualização: o de três dimensões e o modelo de agregação visual. O primeiro pode ser utilizado para analisar aplicações levando-se em conta a topologia da rede dos recursos. A visualização em si é composta por três dimensões, onde duas são usadas para mostrar a topologia e a terceira é usada para representar o tempo. O segundo modelo pode ser usado para analisar aplicações paralelas com uma grande quantidade de processos. Ela explora uma organização hierárquica dos dados de monitoramento e uma técnica de visualização chamada Treemap para representar visualmente a hierarquia. Os dois modelos representam uma nova forma de analisar aplicação paralelas visualmente, uma vez que eles foram concebidos para larga-escala e sistemas distribuídos complexos, como grids.
Palavras-chave: Aplicações paralelas, análise de desempenho, visualização, visualização em 3D, treemap, escalabilidade, grid.

Titre: Quelques Modèles de Visualisation pour l'Analyse des Applications Parallèles.
Résumé: Cette thèse tente de résoudre les problèmes des techniques traditionnelles dans la visualisation du comportement des applications parallèles. L'idée principale est d'exploiter le domaine de la visualisation de l'information et d'appliquer ses concepts dans le cadre de l'analyse des programmes parallèles. La thèse propose deux modèles de visualisation: les trois dimensions et le modèle d'agrégation visuelle. Le premier peut être utilisé pour analyser les programmes parallèles en tenant compte de la topologie du réseau. La visualisation se compose de trois dimensions, où deux sont utilisés pour la representation de la topologie et la troisième est utilisée pour représenter le temps. Le second modèle peut être utilisé pour analyser des applications parallèles comportant un très grand nombre de processus. Ce deuxième modèle exploite une organisation hiérarchique des données et une technique appelée Treemap pour représenter visuellement la hiérarchie.
Mots clés: Applications parallèles, analyse de performance, visualisation, visualisation en 3D, treemap, passage à l'échelle, grille.