Paraver

Version 2.1

Parallel Program Visualization and Analysis tool

TUTORIAL

November 2000

Contents

1 Introduction **1**

1.1 Paraver structure 3

1.2 Paraver tracefile records 3

2 Message Passing Application. NAS LU Benchmark **5**

2.1 What the trace is ? A brief Description. 5

2.1.1 How has the trace file been coded ? 5

2.2 How does it look : Visualization. 7

2.3 A look into details : Zooming. 8

2.4 Timing. 11

2.5 Changing the displayed information : Semantic. 12

2.6 Interested in ultimate detail : Textual display. 13

2.7 Too much information displayed : Filtering. 15

2.7.1 How can I filter communications. 16

2.7.2 How can I filter events. 18

2.8 Measuring things : Analyzer 18

2.8.1 Making a simple analysis 19

2.9 How parallel is our application ? Parallelism profile. 20

2.9.1 How parallel is my application ? 21

3 OpenMP Instrumentation.
NAS BT Benchmark **25**

3.1 What the trace is ? A brief Description. 25

3.1.1 Defined STATES 25

3.1.2 Defined USER EVENTS 26

3.2 How does it look ? Visualization. 30

3.3 A look into details : Zooming. 31

3.4 Interested in ultimate detail : Textual display. 35

3.5 Analyzing the parallel execution 36

3.5.1 Making a simple analysis 36

3.5.2 Parallelism profile 37

3.6 Identifying loop iterations and functions. 39

3.6.1 Identifying loop iterations. 39

3.6.2 Analyzing the five main functions. 41

3.7 Showing benchmark data cache misses. 44

3.7.1 Showing data cache misses profile 45

3.7.2 Showing data cache misses in function of time 47

3.8 Window configuration files supplied for this chapter. 50

4 Hardware counters profile. NAS BT Benchmark **53**
4.1 What is the trace ? A brief Description. 53
4.1.1 Defined USER EVENTS 53
4.2 Visualization of Graduated floating point instructions. 55
4.3 Visualization of data cache misses. 58
4.3.1 Primary data cache misses. 58
4.3.2 Secondary data cache misses. 58
4.3.3 TLB data misses. 59
4.4 Instruction set used by the application 60
4.5 Window configuration files supplied for this chapter. 61

5 Multiprogrammed Executions. Visualization and Analysis **63**
5.1 What the trace is ? A brief Description. 63
5.1.1 Workload description 63
5.1.2 Defined STATES 63
5.1.3 Defined COMMUNICATIONS 64
5.2 How does it look : Visualization. 64
5.3 Study of a single application 66
5.3.1 Selecting one application. 67
5.3.2 Computing the number of process migrations and average execution time in each processor. 68
5.3.3 Parallelism profile for each application during the execution. 68

List of Figures

1.1 Visualization of a MPI application . . . 1
1.2 Visualization of an OpenMP application . . . 2
1.3 Visualization of hardware events profile . . . 2
1.4 Paraver Internal Structure . . . 3

2.1 First Displaying Window. NAS LU BenchMark . . . 8
2.2 Window with Events. NAS LU BenchMark . . . 8
2.3 Zooming the first communication area. NAS LU BenchMark . . . 9
2.4 Zooming. NAS LU BenchMark . . . 10
2.5 Local Orders. NAS LU BenchMark. . . . 10
2.6 Saving the Useful view into a file. NAS LU BenchMark. . . . 11
2.7 Timing utility. NAS LU BenchMark . . . 11
2.8 Semantic window. NAS LU BenchMark . . . 12
2.9 State As If function. NAS LU BenchMark . . . 13
2.10 Equivalence between colors and state values. NAS LU BenchMark . . . 14
2.11 Textual information. NAS LU BenchMark . . . 14
2.12 Textual information with TextMode enabled. NAS LU BenchMark . . . 15
2.13 Filter window. NAS LU BenchMark . . . 15
2.14 Displaying the Physical Communication. NAS LU BenchMark . . . 16
2.15 Communications from processor one to All. NAS LU BenchMark . . . 17
2.16 Communications from processor one to five. NAS LU BenchMark . . . 17
2.17 Communications with tag 2. NAS LU BenchMark . . . 18
2.18 NAS LU Functions. NAS LU BenchMark . . . 19
2.19 First Analysis. NAS LU BenchMark . . . 19
2.20 Profile Visualization. NAS LU BenchMark . . . 20
2.21 Profile Zoom.NAS LU BenchMark . . . 21
2.22 Average number of threads in parallel. NAS LU BenchMark . . . 21
2.23 Percentatge with 7 threads - Semantic. NAS LU BenchMark . . . 22
2.24 Percentatge with seven threads. NAS LU BenchMark . . . 22

3.1 First NAS BT visualization. OpenMP Instrumentation. . . . 30
3.2 Saving the Global view into a file. OpenMP Instrumentation. . . . 31
3.3 Thread Creation. OpenMP Instrumentation. . . . 32
3.4 Body Execution. OpenMP Instrumentation. . . . 33
3.5 Making a zoom to search the reduction. OpenMP Instrumentation. . . . 33
3.6 BT Events View. OpenMP Instrumentation. . . . 34
3.7 BT Reduction lock. OpenMP Instrumentation. . . . 34
3.8 Textual information. OpenMP Instrumentation. . . . 35
3.9 Textual information with labels. OpenMP Instrumentation. . . . 36
3.10 Useful view NAS BT. OpenMP Instrumentation. . . . 37
3.11 Simple analysis. OpenMP Instrumentation. . . . 37
3.12 Parallelism profile view. OpenMP Instrumentation. . . . 38
3.13 Parallelism profile zoom. OpenMP Instrumentation. . . . 38

3.14 Profile analysis. OpenMP Instrumentation. 39
3.15 Identifying loop iterarions. OpenMP Instrumentation. 40
3.16 Identifying loop iterations window. OpenMP Instrumentation. 41
3.17 Selecting function compute_rhs. OpenMP Instrumentation. 42
3.18 compute rhs function. OpenMP Instrumentation. 42
3.19 compute_rhs analysis. OpenMP Instrumentation. 43
3.20 Only one call. OpenMP Instrumentation. 43
3.21 Only one call. OpenMP Instrumentation. 44
3.22 Showing data cache misses profile (I). OpenMP Instrumentation. 45
3.23 Showing data cache misses profile (II). OpenMP Instrumentation. 46
3.24 Data cache misses profile. OpenMP Instrumentation. 47
3.25 Data cache misses profile. OpenMP Instrumentation. 47
3.26 Data cache misses in an iteration. OpenMP Instrumentation. 48
3.27 Showing data cache misses profile in function of time. OpenMP Instrumentation. . 49
3.28 Data cache misses profile in function of time. OpenMP Instrumentation. 49
3.29 Data cache misses profile in function of time. OpenMP Instrumentation. 50
3.30 Data cache misses in function of time in an iteration. OpenMP Instrumentation. . 50

4.1 Working with states in infoPerfex traces. Hardware counters profile. 55
4.2 Selecting the "floating instructions" event (I). Hardware counters profile. 56
4.3 Selecting the "floating instructions" event (II). Hardware counters profile. 57
4.4 Graduated floating point resulting window. Hardware counters profile. 57
4.5 Graduated floating instr. zoomed window. Hardware counters profile. 57
4.6 Primary data misses window. Hardware counters profile. 58
4.7 Primary data misses zoomed window. Hardware counters profile. 59
4.8 Secondary data misses window. Hardware counters profile. 59
4.9 Secondary data misses zoomed window. Hardware counters profile. 59
4.10 TLB data misses window. Hardware counters profile. 59
4.11 TLB data misses zoomed window. Hardware counters profile. 60
4.12 Graduated Loads. Hardware counters profile. 60

5.1 Mapping between colors and applications. Multiprogrammed executions 64
5.2 Processor allocation trace file. Multiprogrammed executions 65
5.3 Processor allocation trace file (no migrations). Multiprogrammed executions 66
5.4 Selecting one application. Multiprogrammed executions 67
5.5 Swim application processor allocation. Multiprogrammed executions 68
5.6 Profile zooms to make the analysis. Multiprogrammed executions 69
5.7 Turb3D profile. Multiprogrammed executions 70
5.8 Hydro2D profile. Multiprogrammed executions 70

List of Tables

2.1 Defined states. NAS LU BenchMark 6
2.2 Function Event Values. NAS LU BenchMark 6
2.3 Global Communication Values. NAS LU BenchMark 7
2.4 Percentatge time in parallel. NAS LU BenchMark 23

3.1 Trace file states. OpenMP Instrumentation. 26
3.2 User events related to NAS BT structure. OpenMP Instrumentation. 27
3.3 Miscellaneous events 27
3.4 Related OpenMP programming model events. 28
3.5 Parallel function events. PARALLEL and PARALLEL DO directives 29
3.6 Hardware counters events 29
3.7 Semantic Value returned by Last Evt Type + (Mod+1). OpenMP Instrumentation. 40
3.8 Time within a function. OpenMP Instrumentation. 43

4.1 Specific Event Types. Hardware counters profile. 54
4.2 Main loop function event types. Hardware counters profile. 54
4.3 Executed instructions. NAS BT BenchMark 61

5.1 Mapping between application and states. Multiprogrammed executions 64

Chapter 1

Introduction

Paraver is a flexible parallel program visualization and analysis tool based on an easy-to-use Motif GUI. Paraver was developed to respond to the basic need to have a qualitative global perception of the application behaviour by visual inspection and then to be able to focus on the detailed quantitative analysis of the problems. Paraver provides a large amount of information on the behaviour of an application. This information directly improves the decisions on whether and where to invert the programming effort to optimize an application. The result is a reduction of the development time as well as the minimization of the hardware resources required for it.

Paraver offers a minimal set of views on a trace. The philosophy behind the design was that different types of views should be suported if they provide qualitatively different analysis types of information. Sometimes global qualitative display of the application behaviour is not sufficient to obtain conclusions on where the problems are or how to improve the code. Detailed numbers are needed to sustain what otherwise are subjective impressions. Through the analyzer module a user can collect stadistics and obtain a detailed analysis.

What can be found in this tutorial ?

The **Paraver tutorial** is composed by four examples where each example shows a specific tracefile type. The first example shows the execution of the NAS LU Application where the communications between threads have been implemented using Message Passing Interface (MPI), this tracefile has been done by **Dimemas** tool, this tool is used to predict the parallel execution on platforms with different caracteristics.

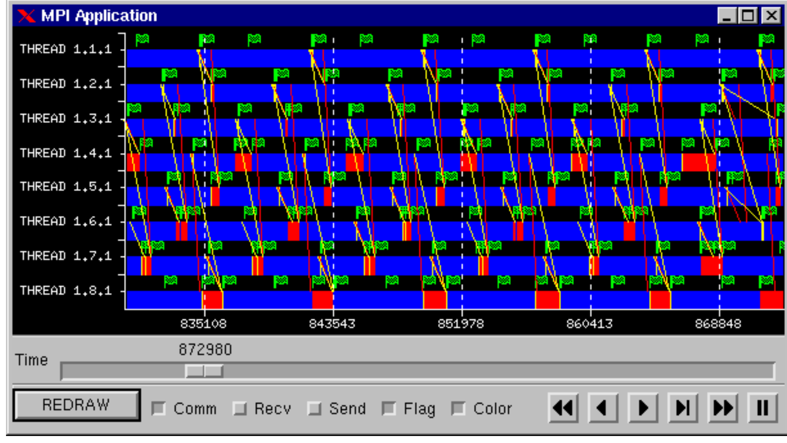


Figure 1.1: Visualization of a MPI application

The second example shows the real execution of a *OpenMP application* generated by a dynamic

instrumentation package. This package take traces from the real execution which are transformed to a Paraver format tracefile. Therefore, using Paraver tools we can visualize and analyze the application behaviour.

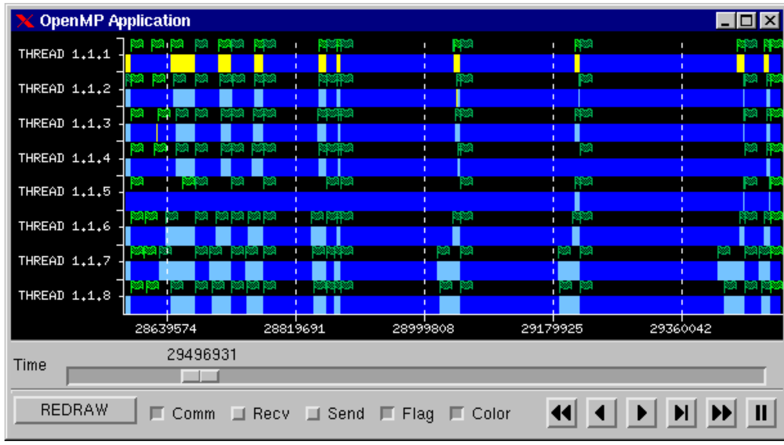


Figure 1.2: Visualization of an OpenMP application

Our third example is composed by a set of tracefiles which contain the *hardware event counters* extracted from the execution of an application. Those traces have been obtained using **infoPerfex** tool which reads during the execution the SGI hardware event counters. We obtain tracefile which tells the profile of primary misses, secondary misses, loads instructions, floating point instructions, ...

Figure 1.3: Visualization of hardware events profile

Finally, the fourth example shows *the processor allocation* of a set of parallel applications during a workload made by the IRIX operating system . The applications request a number of processors greater than the available so the operating system has to distribute the resources between those applications, meanwhile, a tool (**scpus**) collect by sampling each time step which applications are running in each processor. As a result, we obtain a tracefile which has the processor allocation done during the workload. Using Paraver we can visualize and analyze those traces.

Any user can implemented his/her own processor allocation policy onto the machine and analyze its behaviour using **scpus and Paraver tools**.

This tutorial shows how Paraver can be used for several purposes in the area of application tuning. Paraver offers a wide range of utilities and options to study the application behaviour:

- To play with the visualization (communication lines, send and receive icons) and filtering (partners and message filters) parameters, to fix the feature that you think it is most relevant.

- The semantic module could be combined to extract the suited information from the tracefile.
- The zooming utility offers the possibility of magnifying a specific part of the displaying window. Select the ”critical” area to get a the best view of a problem.
- Use the ”global control” buttons to manage some windows at the same time.
- The timing utility offers the possibility of measuring a specific part of the displaying windows.
- The analyzer gives you some quantitative measurements about the displayed tracefile.

The full explanation about each window can be found in **Paraver : Reference Manual** but this tutorial tries to give a basic idea to begin to work with Paraver.
The web side of Paraver is : **http://www.cepba.upc.es/paraver**
Paraver e-mail support : **cepbatools@cepba.upc.es**

1.1 Paraver structure

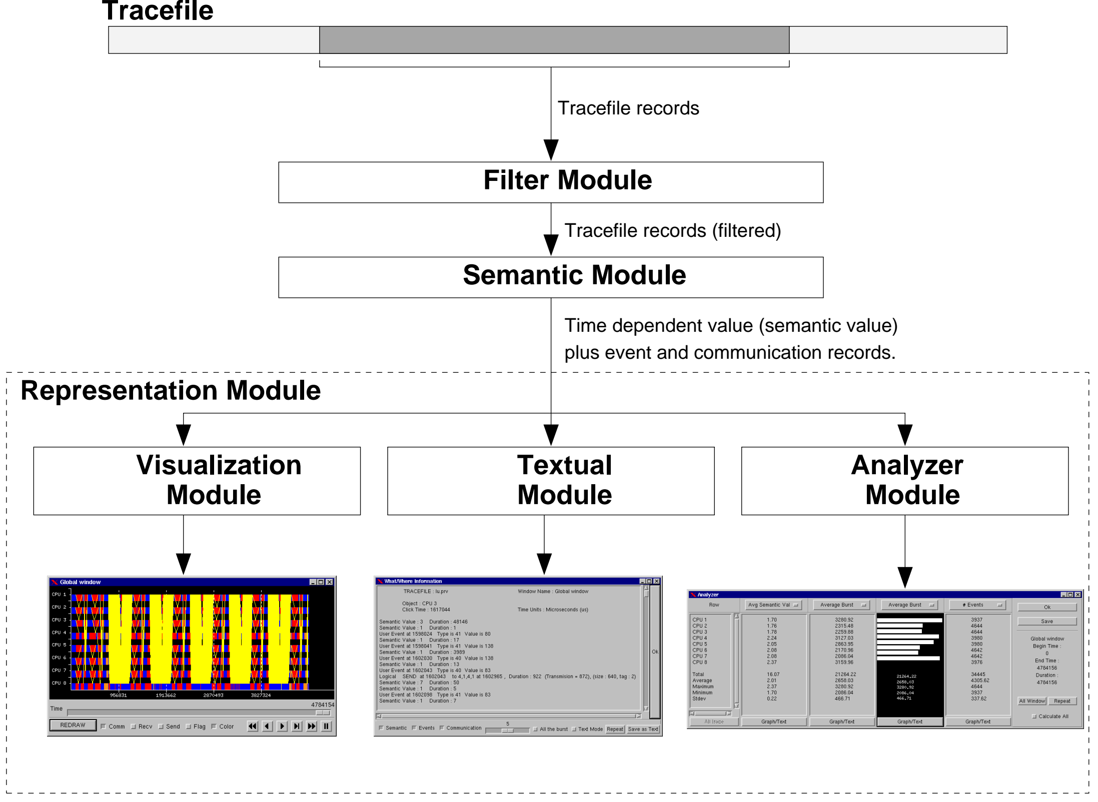


Figure 1.4: Paraver Internal Structure

1.2 Paraver tracefile records

Paraver structure consists of three levels of modules (figure 1.4). First, working onto the tracefile there is the Filter Module. This module gives to the next level a partial view of the tracefile. Second, there is the Semantic Module which receives the tracefile filtered by the previous module and interprets it. The Semantic Module transforms the record traces to time depedent values

which will be passed to the Representation Module. The Semantic Module is the most important level because it extracts and give sense to the record values in the tracefile. The tracefiles have a lot of information that could be extracted and this module selects what will be extracted, this information is called, the **semantics** of the tracefile.

Finally, there is the Representation Module. It receives the time dependent values computed by the semantic module and display it in different ways. The Representation Module drives thus the whole process and offers a graphical display of the tracefile.

The visualization tracefile contains records which describe absolute times at/during which events/activities take place on a run of the parallel code. Each record represents an event or activity associated to one thread in the system. Three basic types of records are defined in Paraver:

State records represent intervals of actual thread status or resource consumption.

Event records represent punctual user defined "events" in the code. These user events are often used to mark the entry and exit of routines or code blocks. They can also be used to flag changes in variable values. Event records include a type and a value.

Communication records represent the logical and physical communication between the sender and the receiver in a single point to point communication. Logical communications correspond to the send/receive primitive invocations by the user program. Physical communication corresponds to the actual message transfer on the communication network.

Chapter 2

Message Passing Application. NAS LU Benchmark

2.1 What the trace is ? A brief Description.

The first example that we are going to see is an execution of the NAS LU application with MPI (Message Passing Interface). For our working example, the execution has been done with eight threads where all those threads execute the five iterations of the application body.

To obtain the Paraver trace file, we used the instrumentation library based on MPI profiling interfaceVAMPIRTRACE[1] and DIMEMAS[2], a set of tools to instrumentate message passing programs and to predict the parallel program behaviour.

First, we used VAMPIRTRACE to obtain the instrumentation of the NAS LU benchmark execution. As a result, we obtain a DIMEMAS trace which allow to simulate and predict the parallel execution on platforms with differents characteristics. From the simulation, DIMEMAS generate a trace file to make the visualization and the analysis with **paraver**.

With DIMEMAS the user can set different parameters of the target machine and simulate the behaviour in different platforms. To obtain our working example, we select a platform connected by an Ethernet with eight monoprocessor nodes where the network bandwidth has been set to 85 Mbytes/s and each node has a *Remote Communication Startup* of 20 microseconds. We map each thread of the application to a different node.

As a result of the simulation in the platform selected, we obtain an input Paraver trace file that we can be visualized and analyzed. The trace file generated is coded by states, events and communications.

2.1.1 How has the trace file been coded ?

The paraver trace files are composed by states, events and communications traces. During this section, we are going to describe how the message passing application (MPI) has been coded by DIMEMAS tool.

Defined STATES

The Table 2.1 shows the states generated that can be found in a trace generated by DIMEMAS tool. Some of them don't appear in our example. For example, the *Waiting for CPU* state only appears when for example two MPI processes are executing in the same uni-processor node; they must share the physical processor.

[1]information about vampirtrace tool can be found at URL: http://www.pallas.de

[2]information about Dimemas tool can be found at URL: http://www.cepba.upc.es/tools/dimemas/dimemas.htm

State Value	Meaning
0	Idle
1	Running
2	Not created
3	Waiting a message
4	Waiting for link
5	Waiting for CPU
6	Waiting for Semaphore
7	Overhead
8	Probe
9	Send Overhead
10	Recv Overhead
11	Disk I/O
12	Not defined
13	Not defined
14	Not defined
15	Disk I/O Block

Table 2.1: Defined states. NAS LU BenchMark

Defined USER EVENTS

Also, in our example we could see different user events, the event type 40 (to mark a function entry/exit), event type 90 (to mark the entry into a global communication), etc ...

- USER EVENT type 40 : Appears when the thread goes into a function and when exits. Is value at the entry is the number of the function where process goes in, at the exit of a function its value is 0. An example of its values is shown in Table 2.2.

Event Value	Name of the function
	... *(MPI functions)*
80	MPI_Recv (MPI)
81	MPI_Rsend (MPI)
82	MPI_Rsend_init (MPI)
	... *(LU functions)*
138	exchange_1 (Communication)
139	exchange_3 (Communication)
140	exchange_4 (Communication)
141	exchange_5 (Communication)
142	exchange_6 (Communication)
143	init_comm (Setup)
144	jacld (Calculation)
145	jacu (Calculation)
146	l2norm (Calculation)
	...
0	End

Table 2.2: Function Event Values. NAS LU BenchMark

Note that the trace file has stored information about when a processor goes into a MPI function or a LU function.

- USER EVENT types 91, 92, 93 : These event types are used to mark when a process is doing the global communication operations (like MPI_Barrier, MPI_Bcast, ...). The event type 91

appears when the process goes into a global operation; the event type 92 appears on the last process that arrives into the global operation (it tells that global operation can begin); and finally, event type 93 appears when global operation has been executed. Their values tells the global operation that is executed in that moment.

Event Value	Global operation
0	MPI_Barrier (MPI)
1	MPI_Bcast (MPI)
2	MPI_Gather (MPI)
3	MPI_Gatherv (MPI)
4	MPI_Scatter (MPI)
5	MPI_Scatterv (MPI)
6	MPI_Allgather (MPI)
7	MPI_Allgatherv (MPI)
8	MPI_Alltoall (MPI)
9	MPI_Alltoallv (MPI)
10	MPI_Reduce (MPI)
11	MPI_Allreduce (MPI)
12	MPI_Reduce_Scatter (MPI)
13	MPI_Scan (MPI)

Table 2.3: Global Communication Values. NAS LU BenchMark

Defined COMMUNICATIONS

Point to point communications have been defined using communication traces. The communications go *from processor X to processor Y* with their *tag* identifier and their number of bytes (*size*).

In a communication trace we can define the logical times (where communication is ordered) and the physical times (when communication is really done). The logical communication limits have been set when processor in going to send the message and when its going to receive. Physical communication limits have been defined when message is really sended and message is really received by the target processor.

Global communications (like broadcast, barriers, ...) have been defined using states where there is a phase to wait the rest of mpi processes, an overhead code and finally a phase to execute the global communication.

2.2 How does it look : Visualization.

First, launch Paraver, two new windows appear, the **Menu Window** and the **Global Controler**. Go to the option menu *Tracefiles/Load Tracefile* and load the trace file called **mpi_nas_lu.prv** that can be found in directory *tutorial_traces/mpi_nas_lu* in tutorial traces package[3].

When the trace file has been loaded, paraver asks for load its paraver configuration file (floating_point_instructions.pcf) which contains information about the user event labels explained in the previous section. Load it by clicking the LOAD button.

To create a window, press the button Visualizer [*v*] in the Global Controler window, then a new window appears to manage all the windows that will be created, press the CREATE button in the Visualizer window and a Displaying Window should appear named **win_1**. In the Displaying Window you can see the axis, the local orders like Play, Pause, Playback, ... To display the trace in the window go to the button Play in the Displaying Window and press it. As a result, you can see the first visualization of the NAS LU Benchmark execution (Figure 2.1).

[3] traces are available in *Documentation tool* section at URL http://www.cepba.upc.es/tools/paraver/paraver.htm

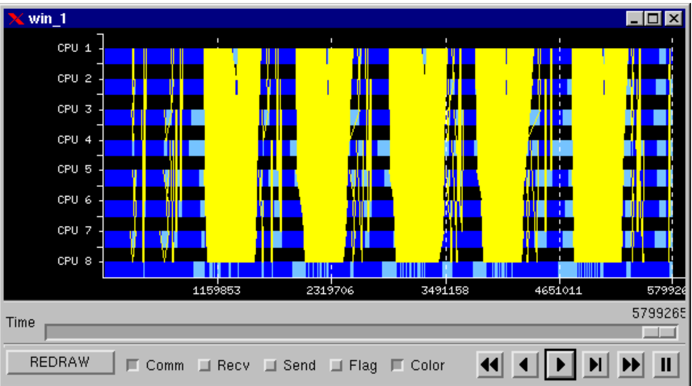


Figure 2.1: First Displaying Window. NAS LU BenchMark

In the **win_1** we see the execution of the NAS LU application. The Y axis represent the eight threads and the X axis shows the time. We can see the five main iterations of the LU execution where the number of communication (yellow lines) is greater than in the other parts. Also, we can see two different states in a thread : the *light blue* which means a non-working state like waiting for a message, overhead, ... and the *dark blue* which means a working state where the thread is doing work.

After creating the window, press the FLAG toggle button and *redraw* the window just pressing the REDRAW button. The drawing area is redisplayed and you can see the events of the trace file represented as green flags (Figure 2.2).

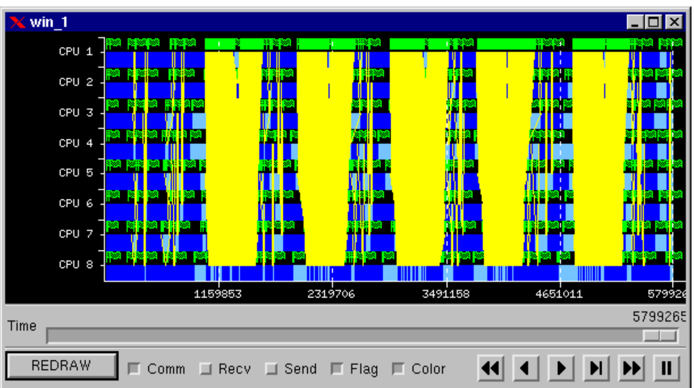


Figure 2.2: Window with Events. NAS LU BenchMark

2.3 A look into details : Zooming.

The previous windows show a global view of the LU execution, but some users can be interested in looking the details of the trace file and not only the global view. The **Zooming** utility offers the possibility of magnifying a specific part of the displaying window to look the details.

To zoom a specific part of the displaying window click the magnifying glass icon in the *Global Controler* window (step **1** in figure 2.3). Then, the **Timing** window will appear in the screen and when the cursor goes into the drawing area on a displaying window it looks like a vertical line to select the specific part to zoom. Click the initial and final points in the displaying window (steps **2** and **3**), in our example we select the points showed in figure 2.3 to obtain a zoomed window which contains the first iteration of LU application.

As you can see in the Visualizer window Paraver create a new window where all the parameters except the X-Scale value has been inherited and where the drawing area displays the selected area (Figure 2.3). This window contains the first iteration of LU application, note the behaviour of the communications, there is a first section where the communications goes from top to bottom and a second section where those communications go from bottom to top. If you zoom any of the other four iterations you will see the same behaviour.

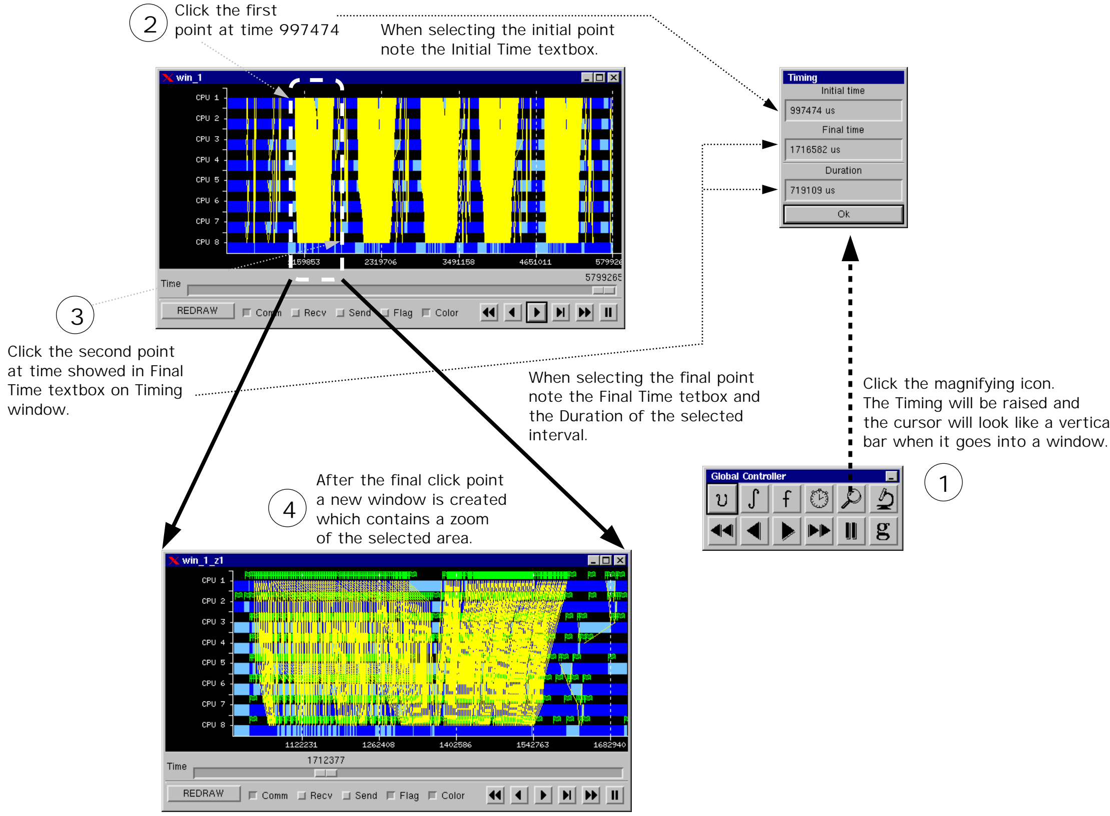


Figure 2.3: Zooming the first communication area. NAS LU BenchMark

The next step is to see how really works one of these iterations zooming in the last created window. Let's go to create a more detailed visualization.

Make two zooms taking as a source window the last zoomed window, the first zoom should be done in the first section of the iteration (when the communications go from top to bottom) and the second in the second section (when the communications go from top to bottom), more or less like figure 2.4. Finally, if you want, you can make another zoom from one of this two windows (like shows figure 2.4). Remember, each time that you want to make a zoom you have to click the **magnifying glass** icon in the *Global Controler* window and select the initial and final points.

Note that successive zooms will show in more detail the trace file, for example, each zoom that has been done, lets to see in more details of the iteration and focus into specific points.

The displaying window works like a tape recorder. On the right-bottom corner there is a set of buttons to manage the drawing area (Figure 2.5).

To display the trace forward press the **Play** button on the window zoom and the drawing area begins to scroll on the trace file. To stop the scroll-on press the Pause button. Also, the trace file can scroll back with the Playback button. With the scale bar and the set of local buttons the trace file can be managed and displayed on different points.

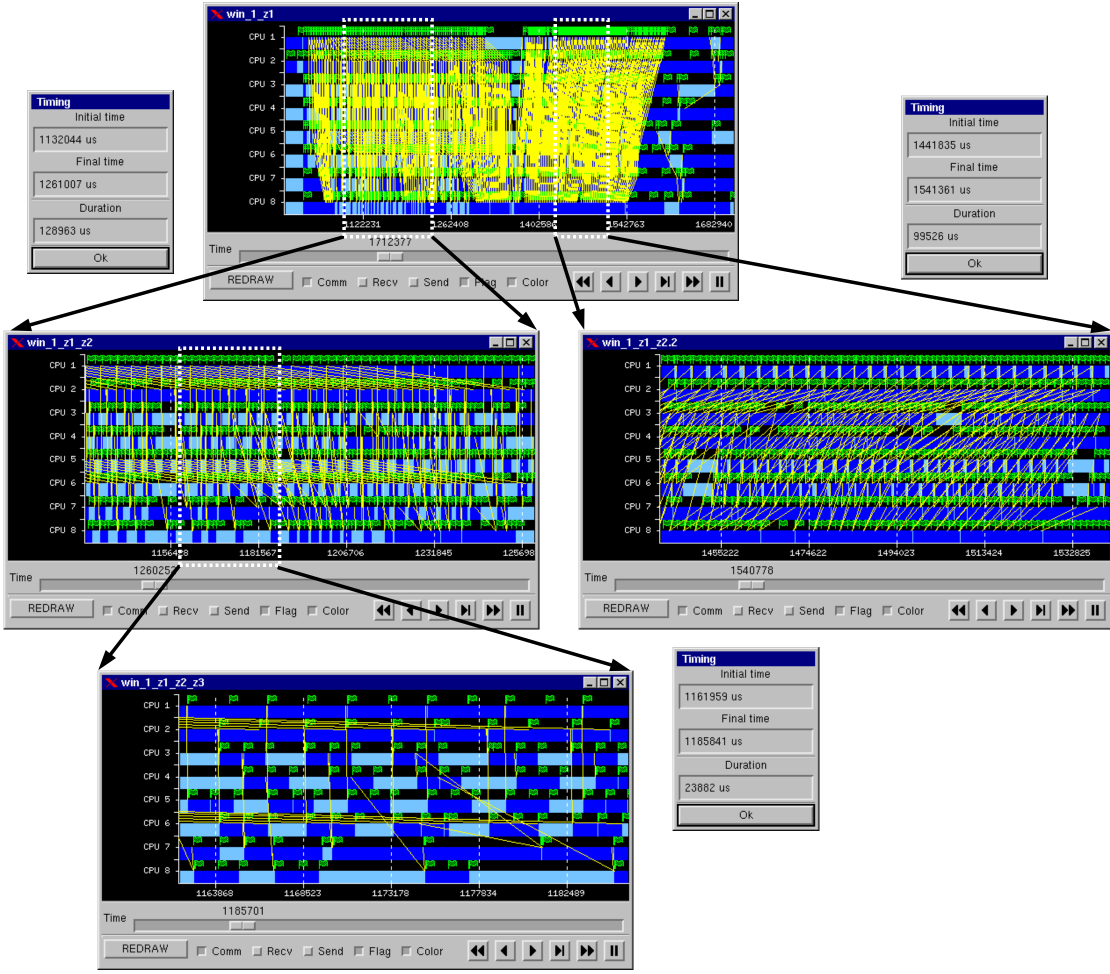


Figure 2.4: Zooming. NAS LU BenchMark

Figure 2.5: Local Orders. NAS LU BenchMark.

Saving a window in a window configuration file

Since one of this windows will be used in next sections (window called **win_1**, we are going to save it in a window configuration file. Before begins to save it, change its name to **Useful view** (go to the visualizer window and select it as the current in the *Window browser* list, type the new name in the *Name* text box, then, click the APPLY button to redraw the window and apply the new name).

To save the window in a window configuration file, select the **Configuration/Save** menu option. It will raise the **Select window** to select the windows that will be saved. Select the **Useful view** window by clicking its name in the *Windows list* and click the SAVE button on the right bottom of the *Select window*; then, write the file name where window will be saved (we recommend the file name *useful_view.cfg* because it will be used in next sections) and click the OK; the window will be saved in a file. Finally, close the **Select window** by clicking its OK button.

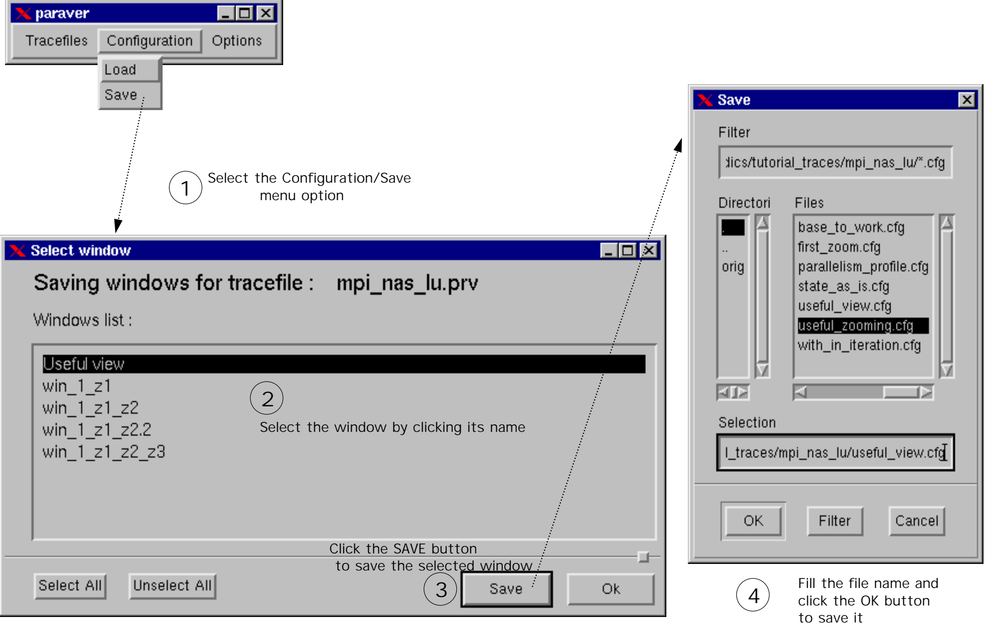


Figure 2.6: Saving the Useful view into a file. NAS LU BenchMark.

2.4 Timing.

To measure a specific part of the window there is the **Timing** utility. This utility works like the **Zooming**. First, click the clock icon in the *Global Controler* window (step **1** in figure 2.7). As in the *Zooming* the **Timing** window is raised and when the cursor goes into the displaying window it looks like a vertical line. Click the initial and final points (steps **2** and **3**) in the displaying window and you will see in the **Timing** window the duration of the selected part.

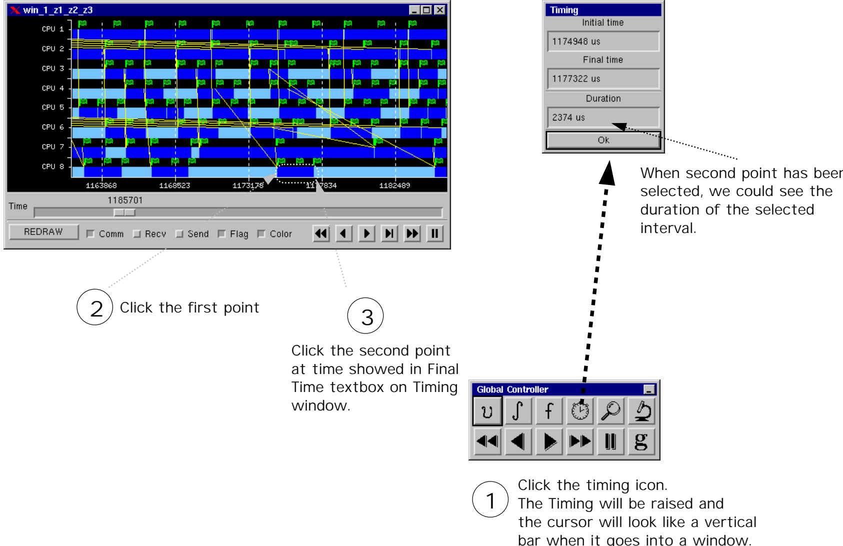


Figure 2.7: Timing utility. NAS LU BenchMark

In the Figure 2.7 we are measuring the third dark blue zone in the CPU number 8. After click the initial and final points the Timing window contains the interval selected and its duration.

2.5 Changing the displayed information : Semantic.

The information displayed in the previous windows is called the *Useful* state view where the working state is painted as *dark blue* and a non-working state is painted as *light blue.*

As we said in the previous sections, the trace file is composed by states, events and communications which not only contains the Useful information and more information can be extracted from the trace file through the *Semantic* module.

The semantic module helps you to interpret the state and event traces through the model process levels.

Click now onto the semantic icon $\boxed{\int}$ on the Global Controler window and the *Semantic Window* will appear (Figure 2.8).

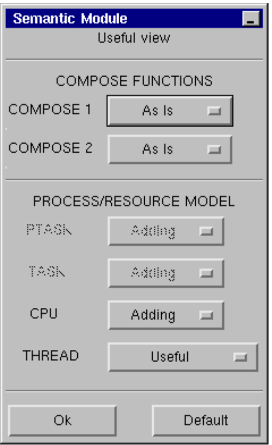


Figure 2.8: Semantic window. NAS LU BenchMark

The name of the current displaying window is shown at the top of the window. Also, in the Semantic window we can see the different levels of the Paraver trace files plus the compose levels that will be explained in the next sections.

At the thread level we can see the *Useful* state view. The Useful state view is used to work with the object activity. Each level has a sub menu to select the function that will be applied. In our example we are working on the CPU level and only are enabled the THREAD and CPU sub menus. The THREAD level functions select what type of information will be extracted from the traces.

First, select the window called *Useful view* in the *Window browser* on the visualizer window. The window name *Useful view* appear like the current one at the top of the semantic window. Then, go to the Semantic window and click the **thread** sub menu; in the pop up menu raised you can see the functions implemented by default at the THREAD level. Select the function STATE AS IS and redraw the window (click the REDRAW button). Also, change its name to **State As Is view** by modifying its name in the Visualizer Module window and pressing the APPLY button.

Now in the displaying window (Figure 2.9) the states are painted with its state value. Repeat the process for the window *win_1_z1* : select it, change its thread level function and change its name to **One iteration**. Note that new states appears in the displaying window, to view the equivalence between colors and states values click the button **Colors** in the *Visualizer window*; this will raise a window where you can see in which colors are painted every state (Figure 2.10). When make a zoom onto a window those values are inherited so the new window has the **State As Is** function selected.

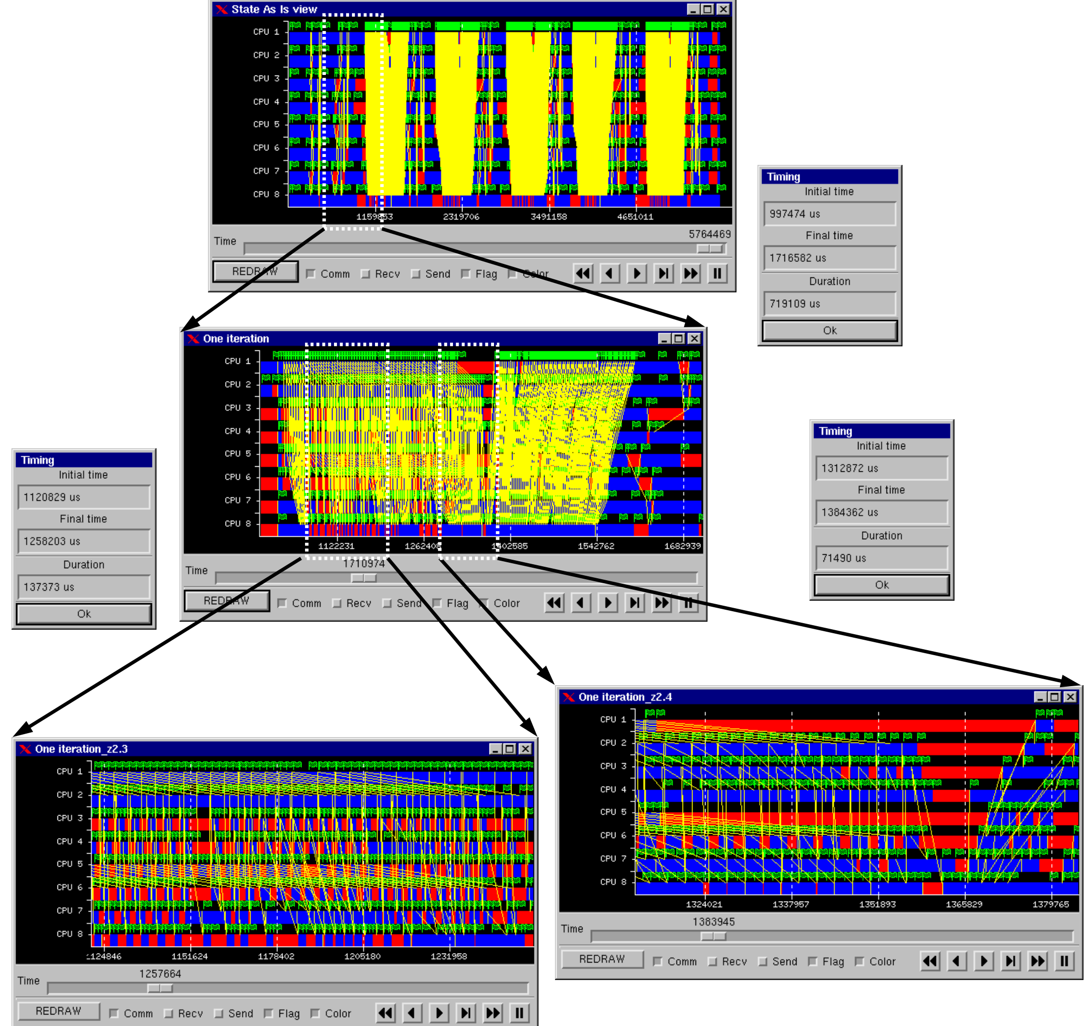


Figure 2.9: State As If function. NAS LU BenchMark

2.6 Interested in ultimate detail : Textual display.

The displaying window offers you a graphical information to understand the application behaviour, but sometimes it is not enough; if you need specific and detailed information about certains point Paraver offers a textual display of the trace file. This utility is called the *What/Where* information and offers a textual display of the information around a selected point.

Click on a row in the drawing area of the displaying window *One iteration* and then you will obtain the textual information around this point. As example, we click on the CPU 1 at time 1172454 microseconds (the first small yellow zone in the CPU number 1) and obtain a textual information around the selected point (Figure 2.11).

As a result, we obtain the information around the selected point, observe that in the textual display there is the information about the semantic value, events and communications. In our example, we are working with states (remember that the function used in the Semantic Module was *State As Is*) so the semantic value refers to state values. For example, the semantic value 1 refers to the state working, the semantic value 7 refers to the state overhead. Also, in the textual display we obtain information about the events and communications around the selected point.

The textual display can offer its information like a numerical view (Figure 2.11) where all the

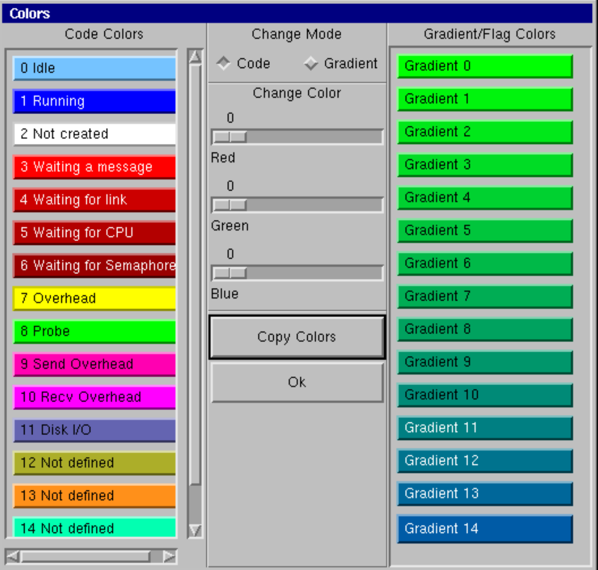


Figure 2.10: Equivalence between colors and state values. NAS LU BenchMark

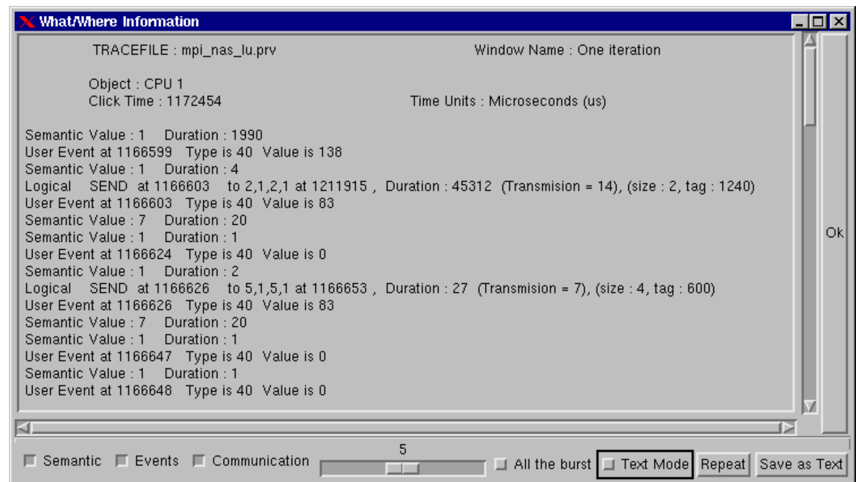


Figure 2.11: Textual information. NAS LU BenchMark

information appear as numbers or it can be viewed as labels, where appear the label related to the number, if it exists. To view the textual display with labels enable the toggle button at the right bottom of the What/Where window called **Text Mode** (see figure 2.12).

In this point we can see in more detail what it is happening. In our selected point, we can see in which functions the thread goes in. For example, the textual display shows how the thread goes first into the function *exchange_1* and while it is within this function, it calls two times to the function *MPI_Send* to send a message. Also, we can see the different states where the thread goes. For example we can see that the communication startup within the call *MPI_Send* is 20 microseconds as we explained in section 2.1.

Note that this utility collects the traces around the selected value; if you are working in a high scale of visualization, the number of lines raises and the clearness is lost. Select an adjusted scale to get the desired level.

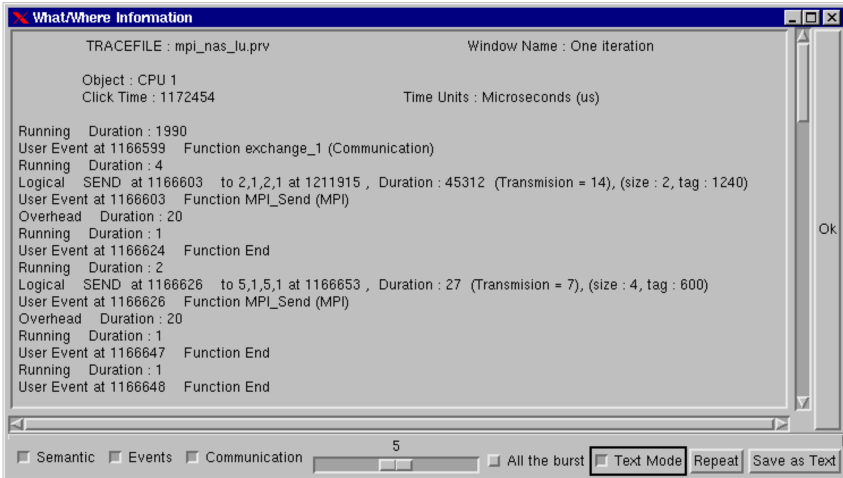


Figure 2.12: Textual information with TextMode enabled. NAS LU BenchMark

2.7 Too much information displayed : Filtering.

Sometimes when there are a lot of events and communications in the displaying window the visualization won't give you information, because the drawing area have so much icons, and communication lines. To avoid this effect, Paraver lets you to filter the trace file to see whatever you want.

Click on the filtering icon *f* on the *Global Controler* window to raise the **Filter** window (Figure 2.13). This window lets you to filter communications and events that won't be displayed.

Figure 2.13: Filter window. NAS LU BenchMark

The name of the current displaying window appear at the top of the window. Note, that there are two main parts, one to filter communications and the other to filter events. The next two points explain how to use those parts.

During this section, we are going to use window called **win_1_z1_z2_z3** created on page 10.

Before begin this section, select the thread semantic function **State As Is** instead of **Useful** function and change its name to **Inside an iteration**; to change it, select the window as the current (by clicking the *Window browser*) change its name in the NAME text box and applying the changes.

Also, save it in a window configuration file called **inside_an_iteration.cfg** (to save it, remember the end of section 2.3). By saving windows in window configuration files, you can shutdown paraver and load the windows the next time to continue with the *Paraver Tutorial*.

2.7.1 How can I filter communications.

Our working example has two types of communications, the logical communications and the physical communications. By default, physical communications are filtered. To see them, select the zoomed window called *Inside an iteration* on the *Window browser*; then, you can see at the top of the filter window the name of the window and now, enable the toggle button PHYSICAL, and *redraw* the zoomed window. After redraw the displaying window you can see the physical communication painted as red lines (Figure 2.14).

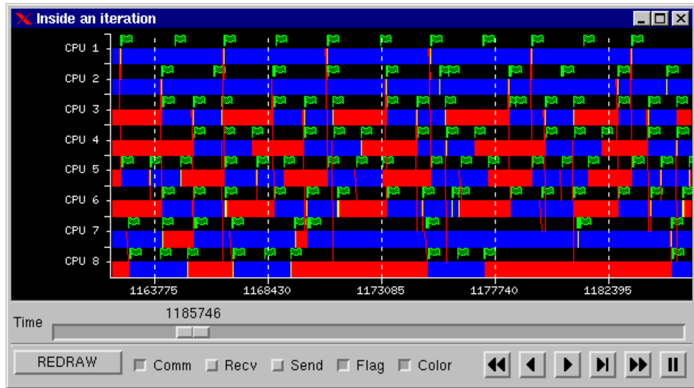


Figure 2.14: Displaying the Physical Communication. NAS LU BenchMark

The logical and physical communications are the two main types of communications. Inside this types the filter window allows filter communications between specific partners or specific size and tags. If no logical and no physical communication are selected, all the communications will be filtered.

How can I display communications between specific partners

To filter communications between specific partners put your attention in the **Partners** area on the **Filter** window.

At the first, we will display the outgoing communications to the processor 1; but before do it, be sure that the current window is the zoomed window *Inside an iteration* (if it is the current you should see his name at the top of the Filter window). Go to the FROM sub menu and select the symbol "=", when this function is selected the text box is enabled. Write in the *Form textbox* the number 1 and redraw the window. Now, in the displaying window you can only see the outgoing communications from processor 1 to the other processors (Figure 2.15).

Now, select the symbol "=" on the TO sub menu and fill in the *To textbox* the processor 5. Then redraw the window and you only will see the communications that goes from processor 1 to processor 5 (Figure 2.16).

Also, the communications can be filtered **From X** *And/Or* **To Y**. As example, we could see the communications that goes out the processor 1 **or** arrives to the processor 5 only selecting the OR toggle button between the FORM and TO lines.

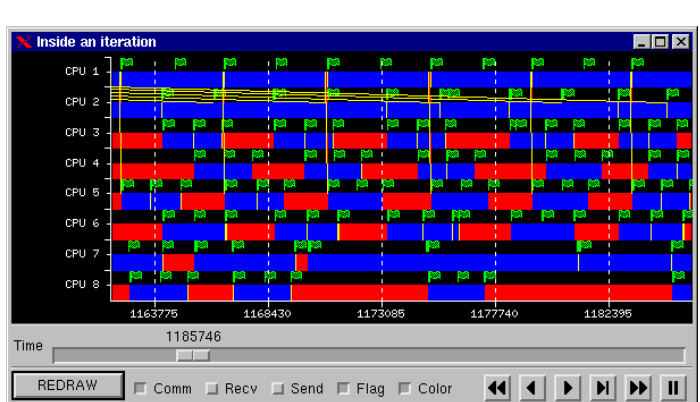
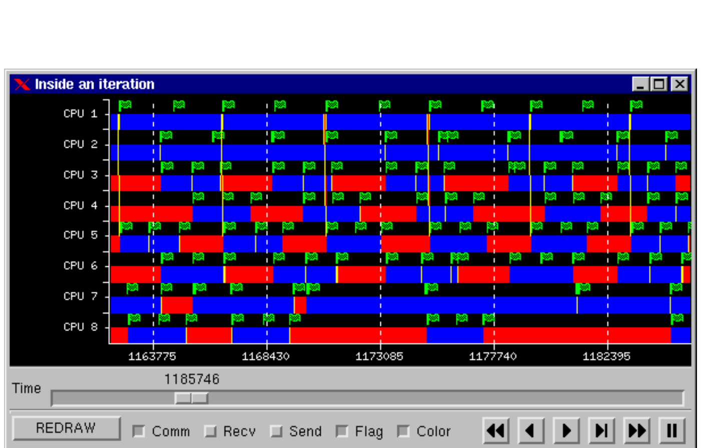


Figure 2.15: Communications from processor one to All. NAS LU BenchMark



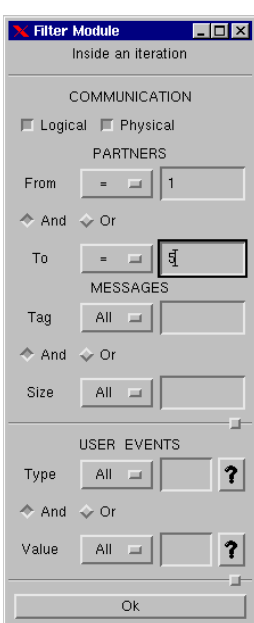


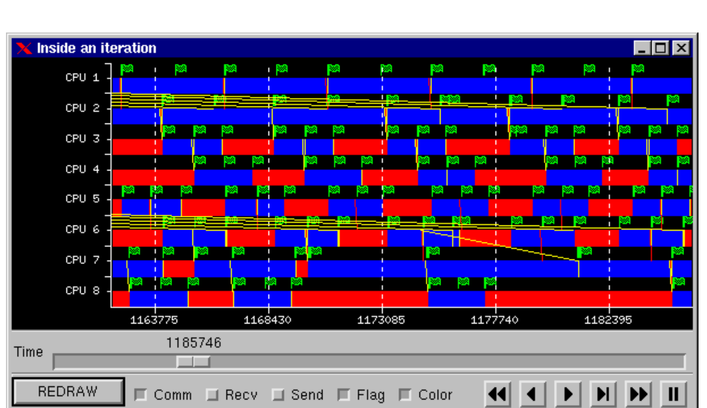
Figure 2.16: Communications from processor one to five. NAS LU BenchMark

How can I display communications with a specific tag and size

The mechanism to filter communications with specific size and tag is similar to the mechanism to select communications between specific partners. First, be sure that there isn't any communication filtered between partners (the From and To sub menus must have the All selected). Don't worry if there is the one and five processors selected in the textboxs, Paraver only uses it if the textbox is enabled.

Now, go to the Tag sub menu, select the symbol "=" and fill the number 2 in the right side textbox; redraw the window and you only see the communications lines which has tag 2 (Figure

2.17).



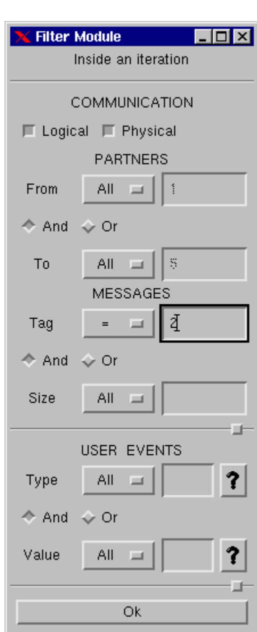


Figure 2.17: Communications with tag 2. NAS LU BenchMark

To select a specific communication size do the same on the size line. Also, we can select a specific size, a size greater or less that a value and a size different than a value. As in the partner selection, you can use the *And/Or* toggles for combine the size and tag selections.

2.7.2 How can I filter events.

Now, we are going to work with events and learn to filter them. First, be sure that all the communications are filtered; to filter all the communications disable the *logical* and *physical* toggle buttons at the top of the window.

In our example trace file there are a lot of events but we are going to focus in the event 40 (Function). To select this type, select in the TYPE sub menu the symbol "=" and put the type. Now, all the events type will be filtered except this type.

Then, we focussed in this event type. Before redrawing the function we are going to filter all the events except the events with the value 133. This event marks the entry of he function called *blts (Calculation)*.

To filter this function go to the VALUE sub menu, select the symbol "=" and fill in the right side textbox the number 133; redraw the window and then, the displaying window will only show the flags with type 40 and value equal to 133 which are the flags that marks the entry of the NAS LU function *blts* (Figure 2.18).

2.8 Measuring things : Analyzer

Paraver offers a analyzer utility to study some features about the application behaviour. Using the Filter and Semantic you can obtain a detailed analysis of the trace file. We are going to start with a simple example to show how use the Analyzer. Later, we are going to compute the average number of processes running in parallel.

The window used in this section can be loaded from a previous saved window configuration file (**useful_view.cfg**). If it exists because you hadn't deleted or you hadn't closed paraver, use it.

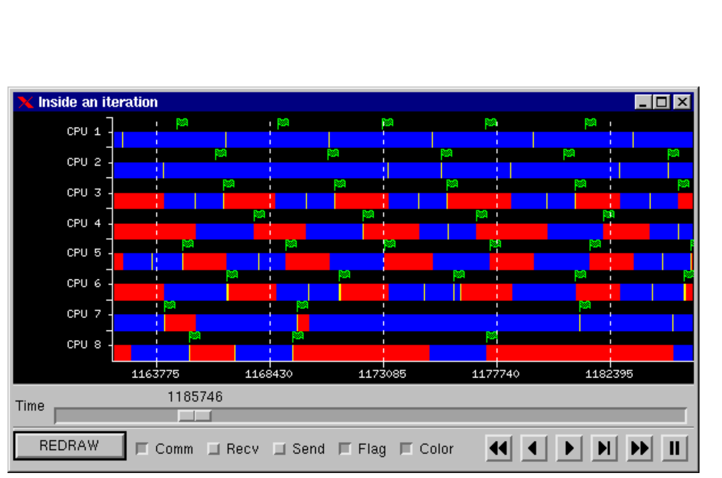


Figure 2.18: NAS LU Functions. NAS LU BenchMark

If not load it from the window configuration file (by selecting the CONFIGURATION/LOAD menu option).

2.8.1 Making a simple analysis

Select the **Useful view** window as the current (the window shows the two default states, the idle state (*light blue*) and the working (*dark blue*).

To compute the analysis click the *Analyzer icon* in the Global Controller window. Two windows will be raised at the same time : the *Timing* window and the *Analyzer* window. Besides, the cursor looks like a little corner when it goes into the displaying window.

At this moment we can analyze all the trace file, clicking the ALL TRACE button in the *Analyzer window*, or a selected area, selecting it in the displaying window. In our first analysis click only the button ALL TRACE on the left corner in the *Analyzer window* to make an analysis of all the trace file, and wait a moment until the analysis will be computed.

The results are written into the *Analyzer* columns (Figure 2.19).

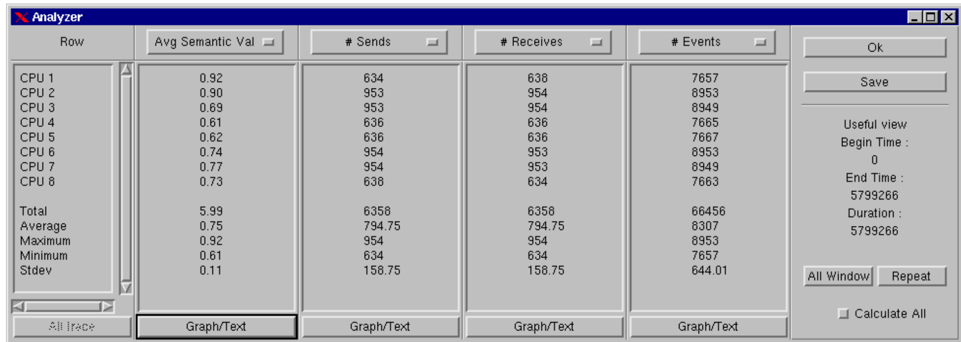


Figure 2.19: First Analysis. NAS LU BenchMark

The first column shows the CPU utilization percentage. This value is obtained from the

integration of CPU burst which state was running during the timing interval prefixed. It could be useful to test the load balancing, in our example the load is well balanced because the *Variance* between all the rows is tiny.

The second and third columns give us the number of sends and receives made during the timing interval prefixed. If the filtering was set with logical communication, then the values are interpreted like the number of logical sends and receives respectively. If the filtering was set with physical communication, then the values are interpreted like the number of physical sends and receives respectively. If the filtering was set with both communication types, then the values are interpreted like the number of logical and physical sends and receives respectively. The logical communication was activated by default.

The fourth column collect the number of user events founded during the timing interval prefixed. If the filtering was not set with some event condition then the analyzer don't care about the user event traces.

The last rows show some computations as the adding, the average, the maximum and the minimum, for each column.

On the right hand of the window you can see the name of the window where the analysis has been done and his limits : initial time, final time and duration.

2.9 How parallel is our application ? Parallelism profile.

An interesting view is the parallelism profiling to see how many processors/tasks are doing work at the same time. This view give us an idea of the parallel behaviour and also, an analysis can be done to study this behaviour.

First, select the window called *Useful view* in the **Window browser**. With this window as the current, click on the PTASK toggle button on the left hand of the Visualizer Window. To show the parallelism profile we have to work at the top level, in our working example this top level is the *Ptask or application* level.

Then, change the Y MAX scale to 8, because we have 8 processors and the maximum value will be 8 threads running at the same time.

Press the **Create** button and a new window will be raised at the Ptask level. Disable the communication lines disabling the toggle button **Comm** at the bottom of the window. By default, the window has enabled the color mode, but when you are working with windows where it's better see them as a time line you can disable the color of the displaying window. Disable the toggle button COLOR at the bottom of the new window and *redraw* the window. This will redraw the axis and now you can click the play button to see the parallel profile of the application.

Without the color we obtain a timing function visualization (Figure 2.20) where the window is showed as a function instead of a color code. Also, change its name to **Parallelism profile**.

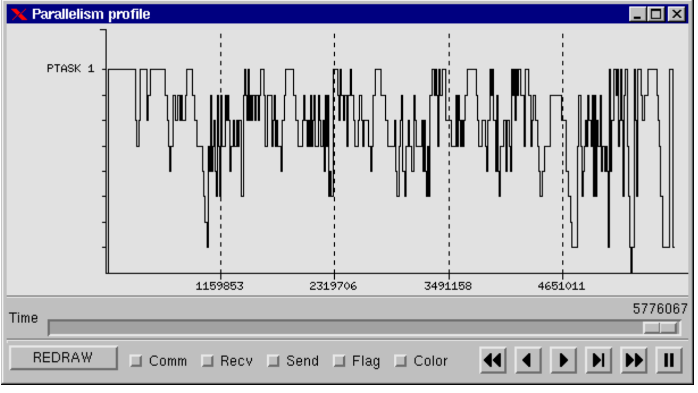


Figure 2.20: Profile Visualization. NAS LU BenchMark

The window shows the parallelism profile of the application. Note the five main iterations where the oΦparallelism is higher and that the behaviour between them is similar. Make a zoom and you can see the parallelism profiling of the application in more detail.

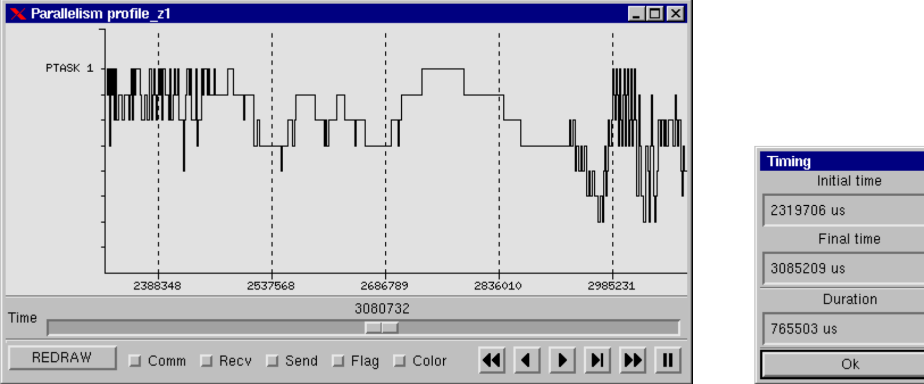


Figure 2.21: Profile Zoom.NAS LU BenchMark

2.9.1 How parallel is my application ?

An easy analysis can be done to resolve this question. We are going to make a small analysis of the parallelism profile to obtain things like : average number of threads running in parallel, how much time we have the maximum parallelism (the 8 threads are doing work at the same time), how much time only one thread is running, ...

Select the profile window called *Parallelism profile*; click the analyzer icon on the Global Controler window and make the analysis for *all the trace*. When the analysis is computed, the function **Avg Semantic Val** gives us the average number of threads that have been running in parallel along the execution.

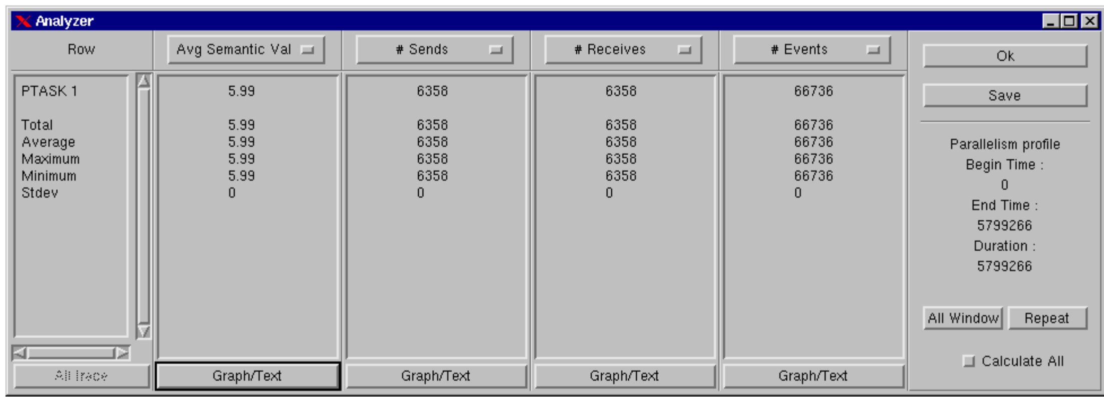


Figure 2.22: Average number of threads in parallel. NAS LU BenchMark

Note that our application has eight threads and the average number of threads running in parallel is **5.99** (figure 2.22).

The next analysis that we are going to do is to compute how much time the application has **n threads** doing work at the same time. We are going to present how to do the analysis when we have seven threads (**n=7**) running in parallel, but the same mechanism can be used for any value of n.

Until now, we have the parallel profile of the application behaviour. To compute this analysis first select in the COMPOSE 2 sub menu the function called **Select Range**, this will raise a

window to fill the range that will be selected, put a 7 in the field *Value Max* and a 7 in the field *Value Min*. This function filter all the values that they aren't between the range, in our example we are only interested in the value 7, which tell us that there are seven processors running in parallel.

In the COMPOSE 1 select the **Sign** function, this will change the seven value to one and we obtain a visualization like the useful where : there are running (or state 1) when the application is working with the seven threads in parallel and a non-working otherwise.

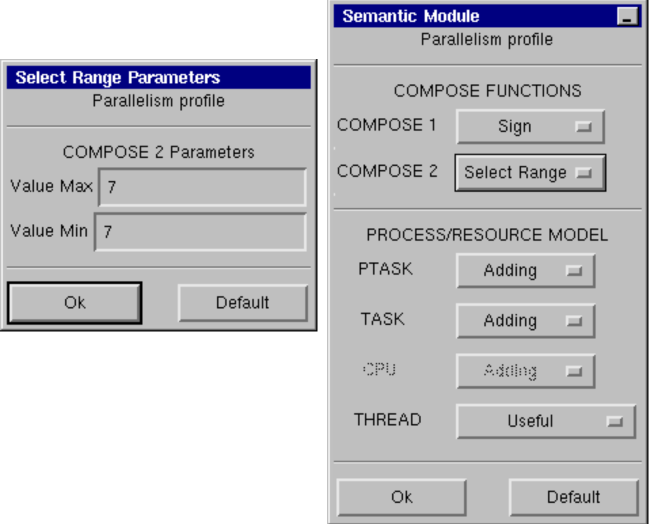


Figure 2.23: Percentatge with 7 threads - Semantic. NAS LU BenchMark

Now, make an analysis for all the trace and when it will be finished in the function *Avg Semantic Val* we obtain the percentatge of the global time that the application is running with seven threads in parallel (Figure 2.24). Also, the *Time with Sem Val* function give us the total time where the application has seven threads running in parallel and functions such as *# Bursts* and *Average burst* give us the number of times that the application has been with seven threads and his average duration time.

Row	Avg Semantic Val	# Sends	# Receives	# Events
PTASK 1	0.20	6358	6358	66456
Total	0.20	6358	6358	66456
Average	0.20	6358	6358	66456
Maximum	0.20	6358	6358	66456
Minimum	0.20	6358	6358	66456
Stdev	0	0	0	0

Figure 2.24: Percentatge with seven threads. NAS LU BenchMark

The analysis can be done for eight, seven, six, ..., to one. We did the analysis only for seven threads but we present the result that should be obtained with the other number of threads running in parallel in the Table 2.4.

The first column shows the number of threads that are running in parallel and the second

column the percentatge of the total time that the application has been with this number.

Number of threads	Percentatge
8	0.25
7	0.20
6	0.20
5	0.18
4	0.07
3	0.05
2	0.02
1	0.04

Table 2.4: Percentatge time in parallel. NAS LU BenchMark

Adding these percentatges we obtain a value of 1 which represents the 100 % of the execution time.

Chapter 3

OpenMP Instrumentation. NAS BT Benchmark

3.1 What the trace is ? A brief Description.

The trace file used in this chapter is the real execution of the NAS BT on a SGI MP environment. The trace file has been obtained through a dynamic instrumentation package that supports a dynamic library interposition mechanism.

During the execution of the benchmark the package collects a trace that corresponds to the real execution of the application and later this trace can be viewed through Paraver. This mechanism can be used with SGI MP programs for which the source code is not available, and it lets us to see its execution behaviour onto the machine. In our example, we have the source code of the NAS BT and we can see how it has been parallelized.

The trace generated by the dynamic instrumentation package contains different states and events which have been recorded during the execution. Our working example has been executed using eight threads, the first thread is the master thread of the application and the others are the slave threads. Those threads help the master in the parallel regions where the parallelism is opened. The execution has been done on a Silicon Graphics Origin 2000 with 64 processors using a cpu set of eight physical processors.

3.1.1 Defined STATES

The table 3.1 shows the states defined in the trace file.

Only bold face states are used by a SGI MP application, the rest are used when a mixed message passing (MPI) and shared memory using OpenMP directives application is traced.

Each state means:

- the **Idle** state is used when threads are waiting for more work (between parallel regions)
- the **Running** state is used when the threads are doing useful work (sequential or parallel application code)
- the **Not created** state is used when threads aren't created. For example, by default in SGI MP environment threads aren't created until the first parallel region, so while thread master is executing the first sequential code the rest of threads are in a not created state.
- the **Blocked** is used when threads are blocked because the number of suggested threads that will take part in the next parallel region has been reduced (in our example, threads haven't been blocked).

State Value	Label
0	**Idle**
1	**Running**
2	**Not created**
3	Waiting a message
4	**Blocked**
5	**Thd. Synchr.**
6	Test/Wait/WaitAll
7	**Sched. and Fork/Join**
8	Probe
9	Blocking Send
10	Immediate Send
11	Immediate Receive
12	**I/O**
13	Group Communication
14	Tracing Disabled

Table 3.1: Trace file states. OpenMP Instrumentation.

- the **Thd. Synchr.** state is used when threads are in a synchronization point, for example this state is used when threads are trying to get/release a lock, or when threads are waiting in a barrier synchronization.
- the **Sched. and Fork/Join** state is used when a thread is doing scheduling code (like scheduling the loops, executing code of the MP library, ...) or when master is waiting at the end of a parallel to join the parallelism so, this state is used to mark when a thread is doing an overhead code.
- the **I/O** state is used when thread is doing a read/write operation.

3.1.2 Defined USER EVENTS

The dynamic instrumentation package has collected some information that have been coded using paraver USER EVENTS. The trace file contains different user events. Working with their event types and values we can extract very different kind of information. Next points describe the trace file user events and their meaning.

NAS BT application structure

Some user events have been used to mark the entry and exit of the main NAS BT functions. These type of user events has been traced manually.

The *NAS BT* benchmark has a main loop that calls five parallel functions. These functions have parallel regions and parallel do directives that tell to the compiler the code that could be executed in parallel. The five functions are repeatedly executed. Before executing this main loop there is an initialization; also, within this initialization phase there is parallel code. The main NAS BT loop looks like :

```
DO I=1, N
    compute_rhs
    x_solve
    y_solve
    z_solve
    add
END DO
```

For example, the function **compute_rhs** is composed by a parallel region (*parallel* directive) with two loops (*do* directive) and four parallel loops (*parallel do* directive), so the parallelism is forked and joined five times (also see *Parallel function events. PARALLEL and PARALLEL DO directives.* on page 28). Other functions like **x_solve**, **y_solve**, **z_solve** and **add** only have one parallel loop.

Event Type	Label	Values	
70000000	x_solve	1	Begin
70000001	y_solve	0	End
70000002	z_solve		
70000003	add		
70000004	exact_rhs		
70000005	compute_rhs		
70000006	initialize		
70000007	error_norm		
70000008	rhs_norm		

Table 3.2: User events related to NAS BT structure. OpenMP Instrumentation.

Table 3.7 shows these event types and their labels. Note that the five main loop functions has been marked with different event types where their values mark the entry and exit to the function. Also, initialization routines have been marked (*initialize* and *exact_rhs*).

We will see this structure in the analysis of our instrumentation example. The NAS BT benchmark could be executed with a different amount of data and iterations. For our working example we used a class known as "class A", which do around 200 iterations to the loop which braces the five functions.

```
Class          =              A
Size           =     64x 64x 64
Iterations     =            200
```

Miscellaneous events

Inside the miscellaneous events group there the I/O events (see table 3.3).

Event Type	Label	Values	
40000001	Application	1	Begin
40000003	Flushing Traces	0	End
40000004	I/O Read		
40000005	I/O Write		
40000011	I/O Size		

Table 3.3: Miscellaneous events

These events mark the read/write operations done by the application and their size. The **I/O Read** and **I/O Write** mark the beginning (*Begin* value) and the ending (*End* value) of the I/O operation, and the **I/O Size** contains as an event value the number of bytes involved in the I/O operation.

The **Flushing Traces** event is used to mark when the dynamic interposition package is flushing traces to disk. In our example, there isn't any flush during the execution because all the trace events have been stored in memory.

Related OpenMP programming model events

The trace file contains user events related to the OpenMP programming model. These event types mark the beginning of parallel code, its synchronization points (lock and barriers) its joining, ...

Table 3.4 shows these event types and their defined values :

Event Type	Label	Values	
60000001	Parallel (OMP)	0	close
		1	Do/Sections (open)
		2	Region (open)
60000002	Worksharing (OMP)	2	End
		3	Begin Do/Sections
		4	Begin Single
60000016	Join (OMP)	1	Begin
60000005	Barrier (OMP)	0	End
60000003	Block (OMP)		
60000007	Lock (OMP)	0	Unlocked status
60000008	Sched. Lock (OMP)	3	Lock
61000000	Reduction lock (address 0x10059100)	5	Unlock
61000001	Reduction lock (address 0x10059180)	6	Locked status

Table 3.4: Related OpenMP programming model events.

- The **Parallel (OMP)** user event type marks the parallel code. Its value tells which type of parallelism is being opened : a parallel region or a parallel do/sections and when it is closed.
- The **Worksharing (OMP)** user event type marks the beginning and the ending of a work sharing construct within a parallel region, this event only can be found within a parallel region and its user event value marks which kind of work (do/sections or a single region) is going to be distributed along the threads that are taking part in the parallel region. For example, in the NAS BT benchmark this event type can be found in the **parallel region** of the **compute_rhs** function.
- The **Join (OMP)** user event type are used to mark in the master thread the joining time at the end of a parallel region.

  We have a **lock type** for each lock found by the instrumentation library to avoid study their behaviour separately. Playing with the event values we could draw when a thread has the mutual exclusion, when is trying to get the lock and when is trying to release it.

  The labels associated to each lock have been generated by the instrumentation package. The names could be : **Sched. Lock (OMP)** which is the lock used by the library, **Unnamed critical lock** used by unnamed critical regions or calls *mp_setlock and mp_unsetlock*, and the **Reduction lock (address 0x...)** used in loops where there is a reduction variable. More locks can be generated by the instrumentation library like named criticals and lock used in OMP calls. But they only appears when working with specific calls and directives.

Parallel function events. PARALLEL and PARALLEL DO directives

The compiler transform the code that has been marked to execute in parallel (parallel regions and parallel do) to a function which can be executed by different threads at same time. The dynamic instrumentation package has marked which function will be executed in each parallel. Their encoding is showed in table 3.5; the values of the event type *Parallel function* is the identifier of each parallel function.

For example, the function **compute_rhs** has five parallel code regions, one parallel region (*__mpregion_compute_rhs_1*) and four parallel loops (*__mpdo_compute_rhs_...*), and for each parallel code region the compiler have generated a function (event values from 9 to 13 on table 3.5). Note that for PARALLEL directives, the function name generated begins with **mpregion**; but for PARALLEL DO directives begins with **mpdo**.

The user event with **End** value is used to mark the end of the parallel region/do.

Event Type	Label	Values	
60000018	Parallel function	1	__mpregion_initialize_1
		2	__mpregion_initialize_2
		3	__mpregion_initialize_3
		4	__mpdo_exact_rhs_1
		5	__mpdo_exact_rhs_2
		6	__mpdo_exact_rhs_3
		7	__mpdo_exact_rhs_4
		8	__mpdo_exact_rhs_5
		9	__mpregion_compute_rhs_1
		10	__mpdo_compute_rhs_1.1
		11	__mpdo_compute_rhs_2
		12	__mpdo_compute_rhs_3
		13	__mpdo_compute_rhs_4
		14	__mpdo_x_solve_1
		15	__mpdo_y_solve_1
		16	__mpdo_z_solve_1
		17	__mpdo_add_1
		18	__mpdo_error_norm_1
		19	__mpdo_rhs_norm_1
		0	End

Table 3.5: Parallel function events. PARALLEL and PARALLEL DO directives

Hardware counter events

The trace file also contains information about cache misses. Since primary cache misses counters and secondary cache misses counter can be taken in the same execution in an Origin 2000, two trace files are provided. The first contains information about primary instruction and data cache misses (NAS_BT_primary_misses.prv) and the second one contains information about secondary instruction and data cache misses (NAS_BT_secondary_misses.prv).

By default, the dynamic instrumentation package has read the hardware counters at the entry and exit of each parallel function (a parallel loop with guided, interleaved or dynamic scheduling can call many times its parallel function) and has coded them by an event type whose value is the number of cache misses occurred between the two reads. Therefore, the user event value at the end of the parallel functions contains the cache misses occurred within them.

Event Type	Label
42000009	Primary instruction cache misses (NAS_BT_primary_misses.prv).
42000010	Secondary instruction cache misses (NAS_BT_secondary_misses.prv).
42000025	Primary data cache misses (NAS_BT_primary_misses.prv).
42000026	Secondary data cache misses (NAS_BT_secondary_misses.prv).

Table 3.6: Hardware counters events

Table 3.6 shows the user event types for each counter and the trace file name where counter has been traced.

During the chapter we only are going to use the **NAS_BT_primary_misses.prv** trace file, but some window configuration files has been supplied to create the next examples for **NAS_BT_secondary_misses.prv** trace file.

3.2 How does it look ? Visualization.

This section will try to show a first view of the trace file. The next sections try to focus the visualization into specific parts and explain how the information could be extracted.

The trace file is composed by states and a lot of user events which have hidden information that could be extracted when combining the different paraver modules.

To create the first view, launch PARAVER and load the trace file which contains the primary misses hardware counter (**NAS_BT_primary_misses.prv**) that can be found in directory *tutorial_traces/omp_nas_bt* in tutorial traces package[1].

When the trace file has been loaded, paraver asks for load its paraver configuration file (NAS_BT_primary_misses.pcf) which contains information about the colors and user event labels explained in the previous section. Load it by clicking the LOAD button.

Now, to create a first window, press the Visualizer button $\boxed{v}$ in the Global Controller window to raise the Visualizer Module window and press the **Create** button. This will create a displaying window named **win_1** where there are painted only the axis. Now, press the Play button on the right bottom corner of the displaying window and we could see the global view of the trace file.

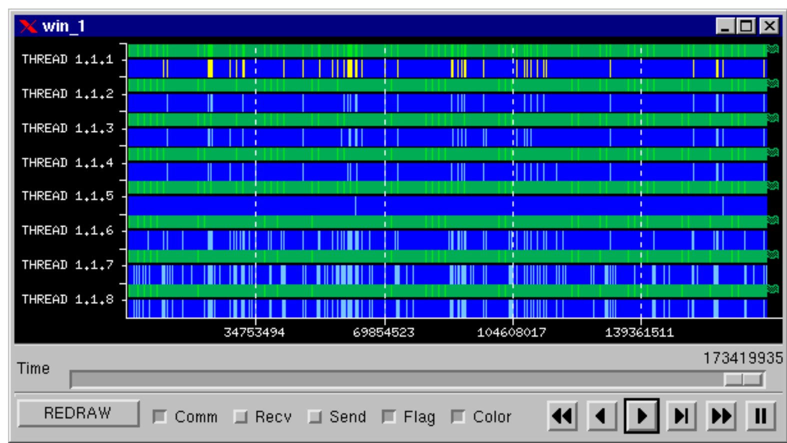


Figure 3.1: First NAS BT visualization. OpenMP Instrumentation.

The visualization (figure 3.1) shows the execution of NAS BT benchmark using 8 threads where thread identified as THREAD 1.1.1 is the master thread of the OpenMP application and the rest are the slaves threads. By default, **State As Is** view is showed for that trace file[2] where dark blue is **Running** state, light blue is the **Idle** state, yellow is the **Sched. and Fork/Join** state, etc ...

Since this window will be used in next sections, we are going to save it in a window configuration file. Before begins to save it, change its name to **Global view** (go to the visualizer window and type the new name in the *Name* text box, then, click the Apply button to redraw the window and apply the new name).

To save the window in a window configuration file, select the **Configuration/Save** menu option. It will raise the **Select window** to select the windows that will be saved. Select the **Global view** window by clicking its name in the *Windows list* and click the SAVE button on the right bottom of the *Select window*; then, write the file name where window will be saved (we recommend the file name *global_view.cfg* because it will be used in next sections) and click the OK; the window will be saved in a file. Finally, close the **Select window** by clicking its OK button.

[1] traces are available in *Documentation tool* section at URL http://www.cepba.upc.es/tools/paraver/paraver.htm

[2] Usually, the default view is the **Useful** view (see the Message Passing Application Example (MPI)) but the paraver configuration file NAS_BT_primary_misses.pcf has changed this property. It has changed the default view to the **State As Is** view.

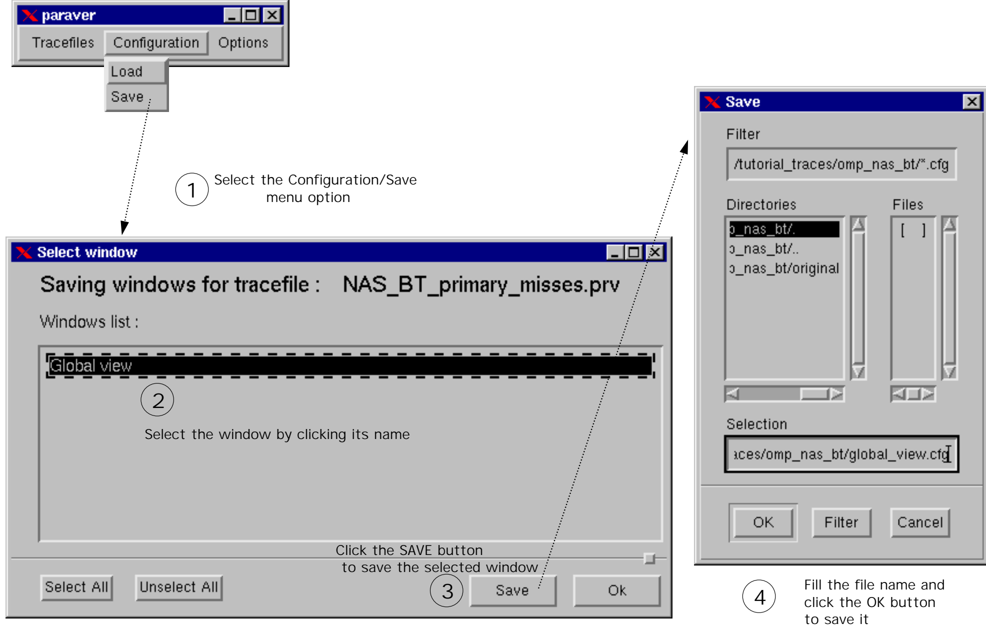


Figure 3.2: Saving the Global view into a file. OpenMP Instrumentation.

3.3 A look into details : Zooming.

The **Zooming** utility lets us to look into the details on we are interested. For example, in this trace file we can see the thread creation, the execution of sequential regions and parallel loops, synchronizations, ... The next points show some aspects that can be found in the trace file, but we suggest that the user should play with the different modules to extract more hidden information that won't be explained in this tutorial.

A look into the thread creation

In this point, we are going to focus our view into the beginning of the application. We are going to create a new window to see the thread creation and the beginning of the first parallel execution.

To zoom the thread creation, click the magnifying glass icon in the Global Controller window (step **1** in figure 3.3). Then, the **Timing** window is raised to show the limits that will be selected; select a small zone at the beginning of the **Global view** window like figure 3.3 (steps **2** and **3**, note the selected limits in the Timing window); when second point has been selected a new window is created where we can see a zoom of the selected interval. In our example, we can see the thread creation plus the beginning of the parallel execution.

Slave threads are created in the first parallelism spawning and when all have been created, the first parallel code begins. They are created only in the first parallelism spawning, the next time where the parallelism is forked these threads will be used. Between the parallel zones where the master thread is executing sequential code, they are in an idle state (wait for work). Sometimes, when this waiting is long the threads block theirself waking up in the next parallel zone.

A look into the execution body

Now, we are going to focus our view into the execution body of the application. As we explained in section 3.1, the application body is composed by five functions that are executed many times, so parallelism is forked and joined many times. The entry and exit to these functions have been marked by USER EVENTS (see table 3.7 on page 40), these events will be used later to make an analysis.

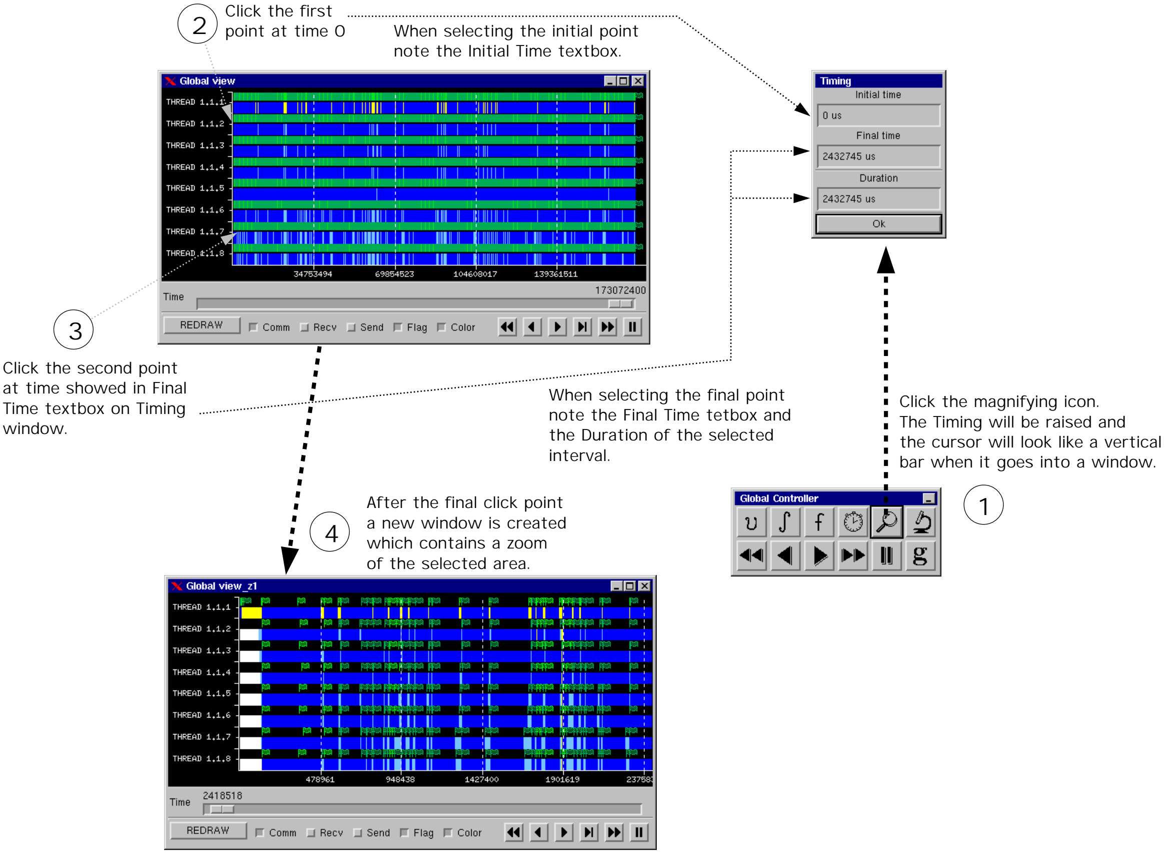


Figure 3.3: Thread Creation. OpenMP Instrumentation.

We are going to make successive zooms to obtain a good visualization of the execution body. First, make a zoom onto the window called **Global view** within the parallel zone; to make this zoom select a region more or less like figure 3.4 to obtain the window called **Global view_z1.2**. This window begins to show a in the execution of the application.

Finally, make another zoom onto the last window called **Global view_z1.2** like figure 3.4 to obtain a new zoomed window called **Global view_z1.2_z2**. This windows shows in more detail the execution body.

Note how the parallelism is opened and forked many times, and between them, there is a bit of idle state. This idle state is a wait for work state that the slaves threads do between two parallelism spawning.

A look into a loop with a reduction

Parallel applications usually use locks to access in a mutual exclusion to shared variables. The dynamic interposition mechanism detect those access and store them in the trace file. Our example only uses two locks to access a reduction variable in mutual exclusion that has been generated by the compiler, but other applications can use many locks and could have behaviour problems.

Our goal is shown how the zooming utility lets to focus our visualization into specific details of the trace file.

First, make a zoom on displaying window called **Global view_z1.2_z2** like figure 3.5 (we are selecting the ending and beginning of a parallel zone). In the new window we can see the finalization of a parallel loop, note that thread master finish its work (ending of dark blue color) and remains in a **Scheduling and Fork/Join** state (yellow color) until the last thread finishes.

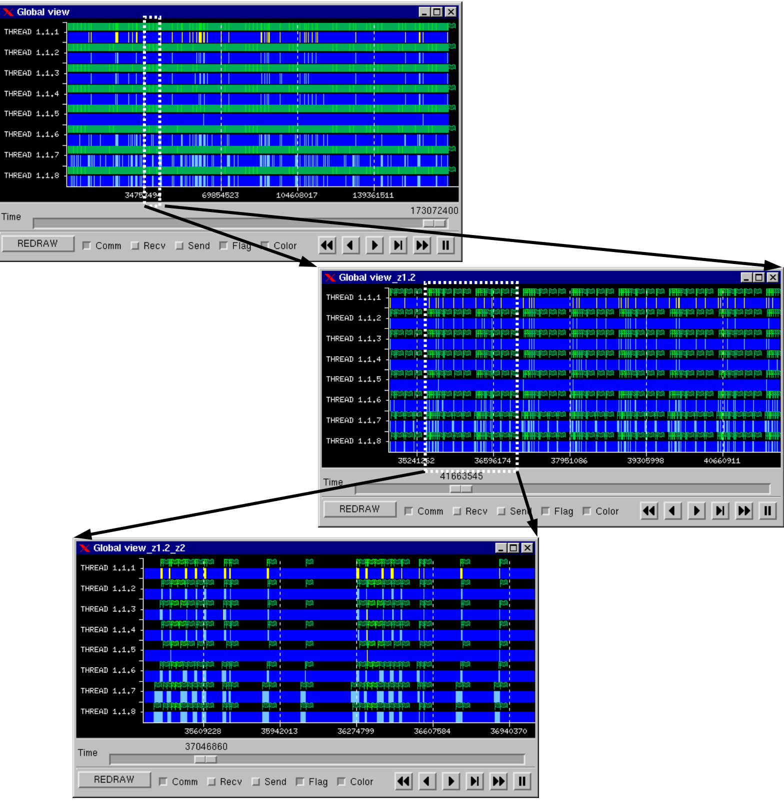


Figure 3.4: Body Execution. OpenMP Instrumentation.

When threads finishes their work, they changes their states to the idle state (waiting for more work) until the next parallelism spawning.

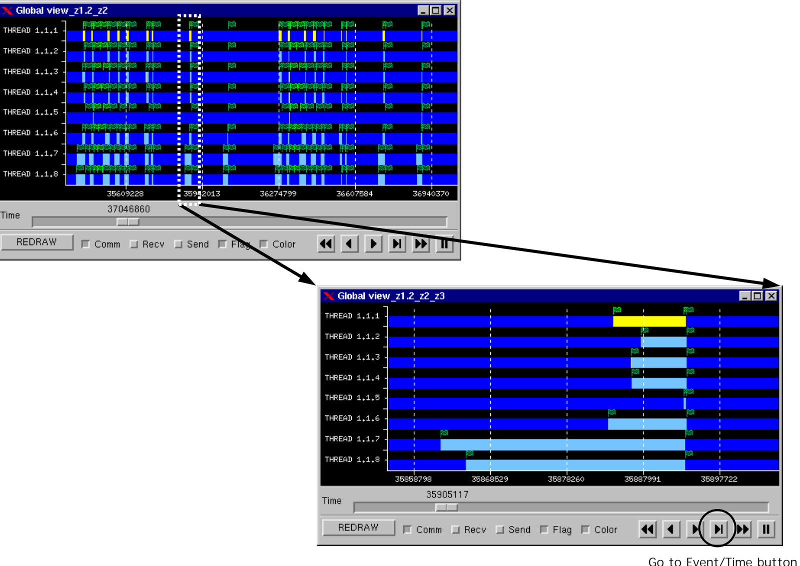


Figure 3.5: Making a zoom to search the reduction. OpenMP Instrumentation.

Now, we have to find one of the reduction lock variables. The instrumentation library has

marked each lock as a different event type. To show their types, click on the **Events button** in the *Visualizer Module* window. It will raise a window (**Events window**) with a list of all defined events types and values (figure 3.6). Search the reduction event types (also see table 3.4 on page 28) by scrolling the event types list.

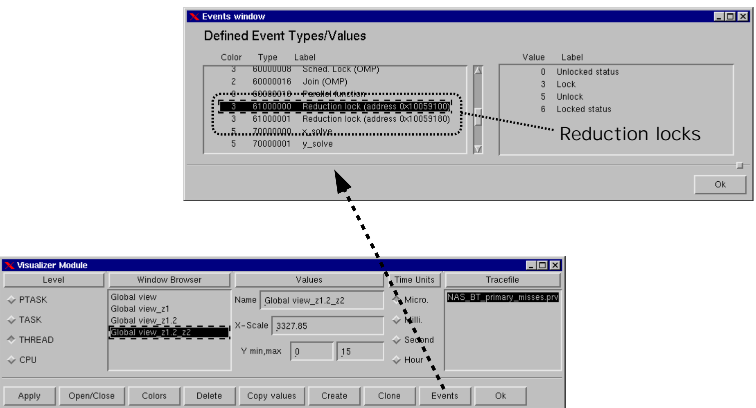


Figure 3.6: BT Events View. OpenMP Instrumentation.

The labels for each event type have been defined in the paraver configuration file that has been loaded in trace file loading.

The current zoomed loop doesn't have any reduction variable. To go to a loop which has one, press the **Go to Event/Time** button in the window called **Global view_z1.2_z2_z3** which contains the end of a parallel loop. This will raise a window to fill the event type that you want to search (figure 3.7); fill the event type 61000000 in the Type text box and press the Go Type button, the **Global view_z1.2_z2_z3** goes to to the first occurrence of this event type.

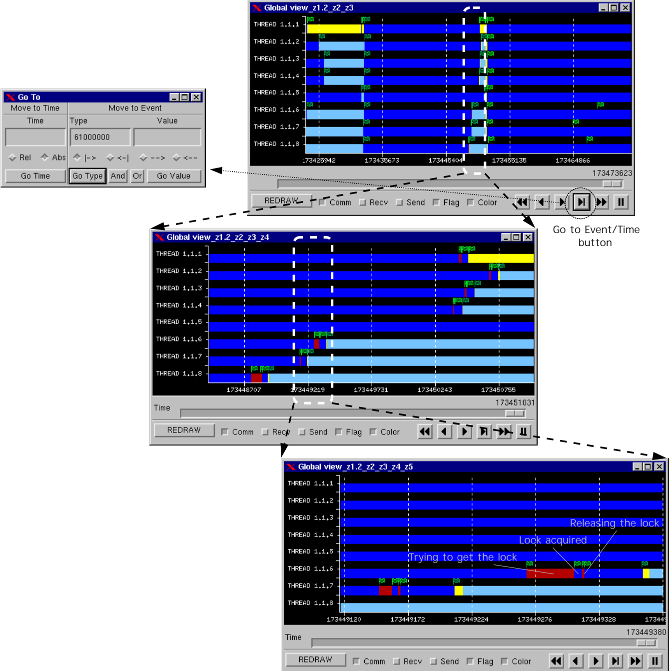


Figure 3.7: BT Reduction lock. OpenMP Instrumentation.

The figure 3.7 shows where is the reduction lock variable, making a zoom of this region we can

see in more detail how does it work.

Note how the lock has been coded (use the textual display to see the flag values). When a thread is trying to get the mutual exclusion and when it tries to release the lock are painted as a **Thd. Synchronization**, the dark blue between them is the execution in mutual exclusion. Each point is marked with flags, where the value tells what it trying to do (see table 3.4 on page 28).

The **Zooming** utility allows us to see this level of detail. Because it is a visualization of the real execution onto the machine, Paraver lets us view and detect the general and the specific problems. A lot of problems could be detected by visual inspection, for example unload balancing, too much synchronization with locks, ... the user has to decide what he/she want to see and understand what it means.

3.4 Interested in ultimate detail : Textual display.

We have seen the graphical visualization of the trace file; but through the textual display we can obtain specific textual information on certain points. For example, click in the parallel zone onto the thread master row in the window called **Global view**. We obtain a textual information around that point (Figure 3.8) which describes what was happening in that moment.

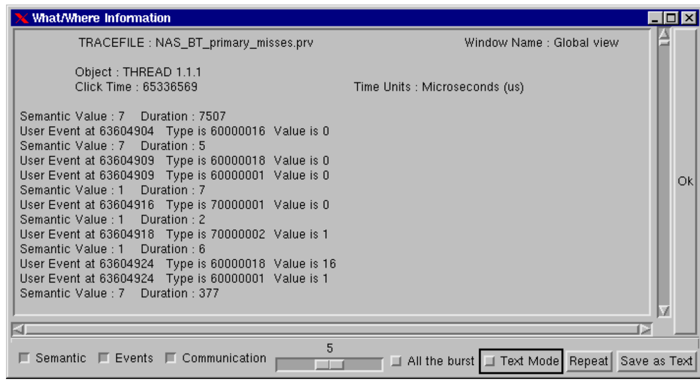


Figure 3.8: Textual information. OpenMP Instrumentation.

In this textual information we can see the USER EVENTS and the semantic value bursts around that point. The semantic value bursts are the states values because we are working in a **State As Is** view where the semantic values returned by the Semantic Module are the *State As Is*. Now, enable the **Text Mode** option (enabling the toggle button *Text Mode* on the right bottom of the What/Where window), the numbers will be shown as labels[3].

The new textual information (Figure 3.9) has the labels in *text mode* which give us more understable information. You can see what is happening around your clicked point, for example, in our clicked point the master thread (THREAD 1.1.1) goes out the function **y_solve** and after a small running state goes into the function **z_solve** where spawns the parallelism in the parallel loop of **z_solve** function (*__mpdo_z_solve_1*). Between those flags you can see the states where the master thread has been and their duration.

According to the scale where the window is working, it could display too many records. Working in a lower scale, the number of traces around the selected point will be less than working in a high scale, therefore, work in a desired level of detail to obtain the desired number of records.

[3]Those labels have been defined in the paraver configuration file

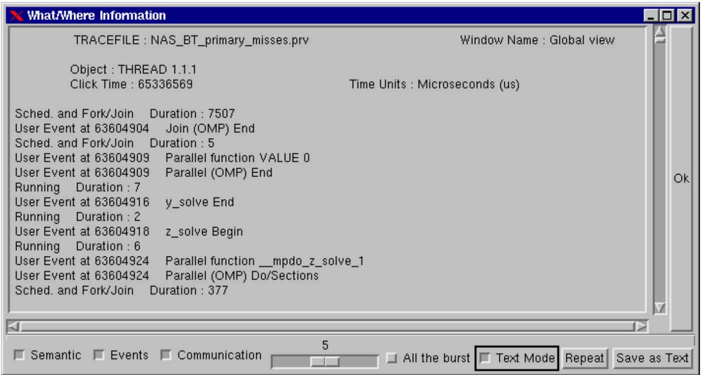


Figure 3.9: Textual information with labels. OpenMP Instrumentation.

3.5 Analyzing the parallel execution

The trace file shown in this chapter contains the real execution of the NAS BT application onto the machine and through the **Analyzer** module we could study its behaviour.

Trace files with more information could be generated using the dynamic interposition, but in our example we only want to see the application behaviour, so we generated a simple trace file.

Before begin the analysis, delete all the created windows or save them in a window configuration file and delete them.

To delete a window close it by clicking its left or right corner, or select it like the current one in the **Window browser** list and press the DELETE button. After delete all the windows, we are going to load the **Global view** window that has bee saved in section **How does it look ? Visualization** on 30. This window will be the base window for next sections.

To load it, go to the *Configuration/Load* menu option; then, select the **global_view.cfg** file name in the file selection box and press the OK button. The **Global view** window will be created.

3.5.1 Making a simple analysis

First, we are going to compute the percentage of time that a thread has been working along the execution. This will give us an idea of load balancing, because if those percentages are very different it could mean that some threads have more work than others because in our example all the threads have taken part in all the parallelism spawning.

The easiest way to make this analysis is to take the window called **Global view**, clone it by clicking the CLONE button in the *Visualizer Module* window. Then, go to the **Semantic Module** and in the THREAD level select the **Useful function** to obtain an useful view of the trace file. Also, rename the window **Global view_c1** to **Useful view** name by changing its name in Name text box in *Visualizer Module* window and apply the changes by clicking the APPLY button. window will be redrawn using the new semantic function and the new name will be applied. Now, all the states except the running state are showed as idle state.

Make an **All trace** analysis for this new window to obtain these percentages (to make the *all trace* analysis, you have to click the analyzer icon in the *Global Controller* window and then click the **All trace** button in the *Analyzer window*).

Figure 3.11 shows the percentage of running state in each processor, note that those values are a number less to 1.0, and the variance between them is about 0.08 (Stdev value at the bottom of the Avg Semantic Val column). This means that load is well balanced because all threads do more or less the same amount of work along the application.

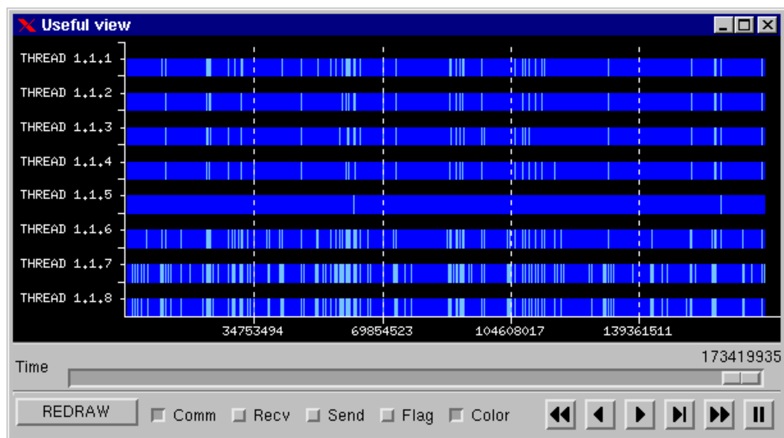
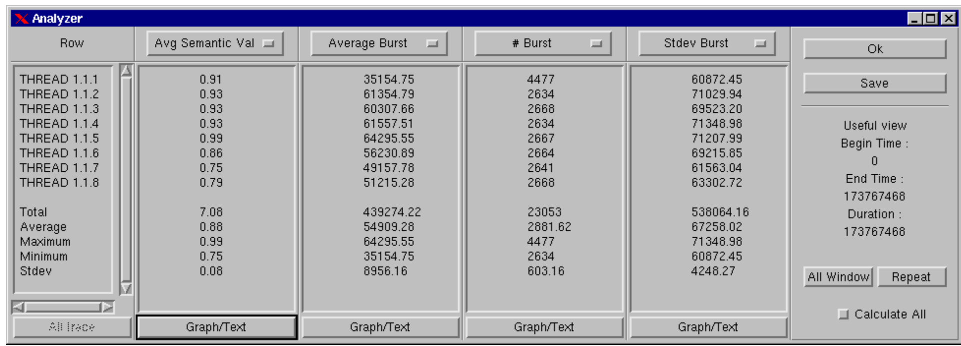


Figure 3.10: Useful view NAS BT. OpenMP Instrumentation.



Row	Avg Semantic Val	Average Burst	# Burst	Stdev Burst
THREAD 1.1.1	0.91	35154.75	4477	60872.45
THREAD 1.1.2	0.93	61354.79	2634	71029.94
THREAD 1.1.3	0.93	60307.66	2668	69523.20
THREAD 1.1.4	0.93	61557.51	2634	71348.98
THREAD 1.1.5	0.99	64295.55	2667	71207.99
THREAD 1.1.6	0.86	56230.89	2664	69215.85
THREAD 1.1.7	0.75	49157.78	2641	61563.04
THREAD 1.1.8	0.79	51215.28	2668	63302.72
Total	7.08	439274.22	23053	538064.16
Average	0.88	54909.28	2881.62	67258.02
Maximum	0.99	64295.55	4477	71348.98
Minimum	0.75	35154.75	2634	60872.45
Stdev	0.08	8956.16	603.16	4248.27

Figure 3.11: Simple analysis. OpenMP Instrumentation.

3.5.2 Parallelism profile

As the in previous example (see Message Passing Application (MPI) NAS LU Benchmark on chapter 2), an interesting view is the parallelism profile of the application. It let us to see and compute how many processors are working in parallel at the same time. A poor parallelism value could mean a bad parallelization of the application but a big value could mean a good load balance. This value has been computed in the previous analysis (see the Total value on Avg Semantic Val in figure 3.11), there was an average of 7.08 threads executing work at same time, but we want to show how the same value can be computed using a different view.

Select the window called **Useful view** like the current one in the **Window browser**, clone it to obtain a cloned window called **Useful view_c1**. Then, disable the **Flag** button on the displaying window to not paint the flag icons.

To see the profile, we have to change the object level representation for this window to the **PTASK** level. To change it, click on the PTASK toggle button on the left of the Visualizer Module (be sure that window **Useful view_c1** is selected like the current one), disable the color mode of the displaying window to obtain a non color visualization, and finally, change the **Y max** scale to 8 because it will be the maximum profile value and its name to **Parallelism profile**[4].

Apply the changes (click the **Apply** button) and the window will be redrawn with the execution profile (figure 3.12).

This window shows the parallelism profile of the application. PTASK level adds the values

[4]To capture the windows we have changed the background and foreground (using the option OPTIONS/SYSTEM COLORS). If you work normally with Paraver without changing any color, foreground will be the white color and background will be the black color.

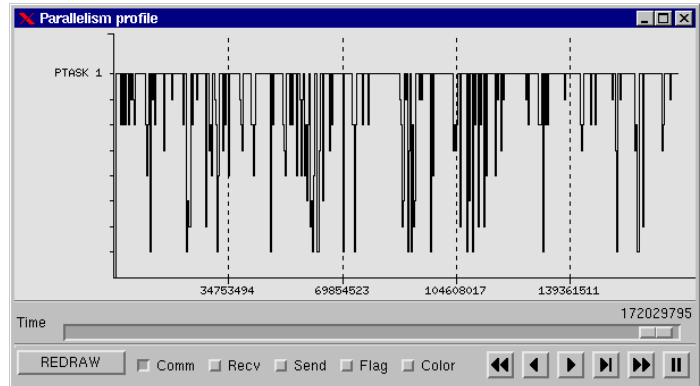


Figure 3.12: Parallelism profile view. OpenMP Instrumentation.

returned from the previous level, the level adds all the values from the useful view where the PTASK level adds all the threads that are in a working state (value 1). Note that during the execution, the parallelism value is high. Make a zoom to see it in detail.

Also, copy the window limits of the parallelism profile zoomed window (**Parallelism profile_z1**) to the **Useful view** window (to copy them, select the Parallelism profile_z1 window like the current, press the COPY VALUES button in the Visualizer Module window, and select the source window (Useful view) by clicking its name in the *Window browser*; the limits will be copied) to compare the two windows (figure 3.13).

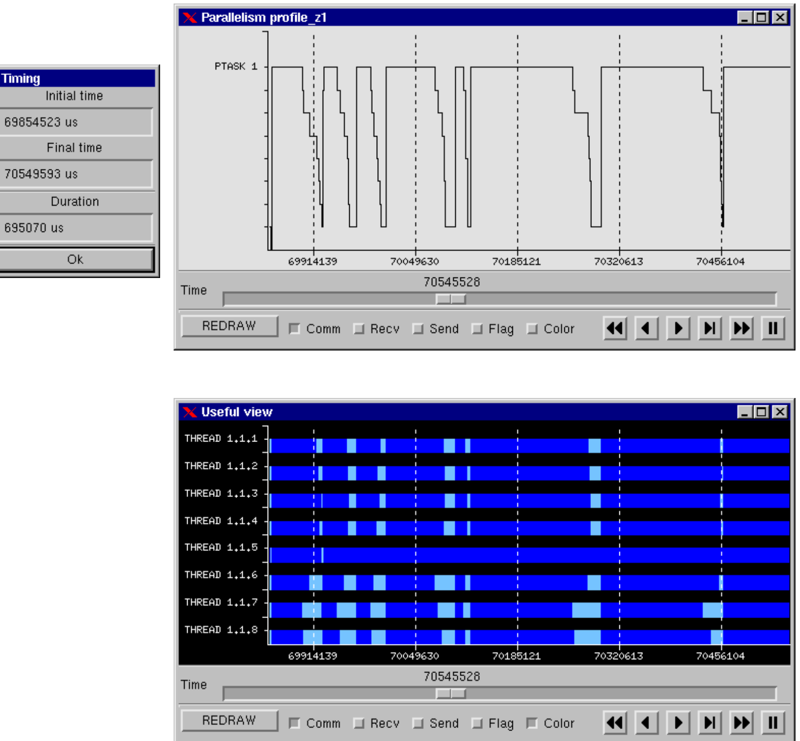


Figure 3.13: Parallelism profile zoom. OpenMP Instrumentation.

Figure 3.13 is a zoom made in the execution body of the application. Note how between maximum parallelism zones (8 threads doing work at the same time) the value goes to one until the last thread finishes the work.

Using the **Analyzer Module** we can compute the average number of threads doing work in parallel only making an *All trace* analysis for the **Parallelism profile** window. The result is showed in the **Avg Semantic Val** function (figure 3.14).

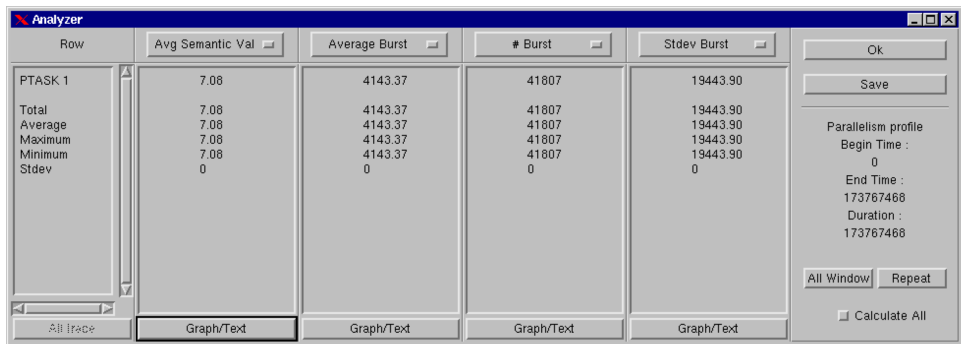


Figure 3.14: Profile analysis. OpenMP Instrumentation.

Note, that the result is 7.08. The maximum or utopic value is 8 because we have only 8 threads but in parallel applications this value can't be obtained because ever there are synchronizations, overhead code, bad cache behaviour, latency to access the memory, ...

Before proceed with next points, save those created windows in a window configuration file and delete them. Next points will load previous saved configuration files.

3.6 Identifying loop iterations and functions.

As we explained in section 3.1 the trace file has information about when the thread goes into one of the five main functions and when goes out. This information has been recorded for the master thread because the parallelism is spawned within this functions, so only the master thread goes into this functions. This information has been encoded by event types where there is a event type for each function, and their values can be 1 (entry to the function) and 0 (exit).

The next points explain how different modules can be combined to identify the loop iterations and each function.

3.6.1 Identifying loop iterations.

The NAS BT benchmark has a main loop that calls five functions (see **What the trace is ? A brief description.** on page 25). These five functions are repeatedly executed.

The dynamic instrumentation package has traced the entries and exits to each of these five functions and coded them by USER EVENTS (see table 3.7 on page 40). The entries and exits to these functions only have been traced by the master thread (parallelism is spawned within those functions) so only THREAD 1.1.1 has these USER EVENTS.

We are going to create a window where entries and exits to these five functions will be showed with different colors, so iterations could be detected.

First, load the **global_view.cfg** window configuration file to load the previous saved window. Clone it to obtain the window called **Global view_c1** (remember that to clone a window you have to press the CLONE button in the Visualizer Module window).

To obtain the desired visualization we have to be in mind how entries/exits has been coded (see table 3.7 on page 40). Each function has been coded by an user event type. There is an events type traced at the end of the function (with value 1) and other at the exit (with value 0); the interval between functions (sequential code between parallel regions) is insignificant because it is very small.

We have to solve the next question : **How different modules can be combined to obtain a visualization where functions will be painted using different colors ?**

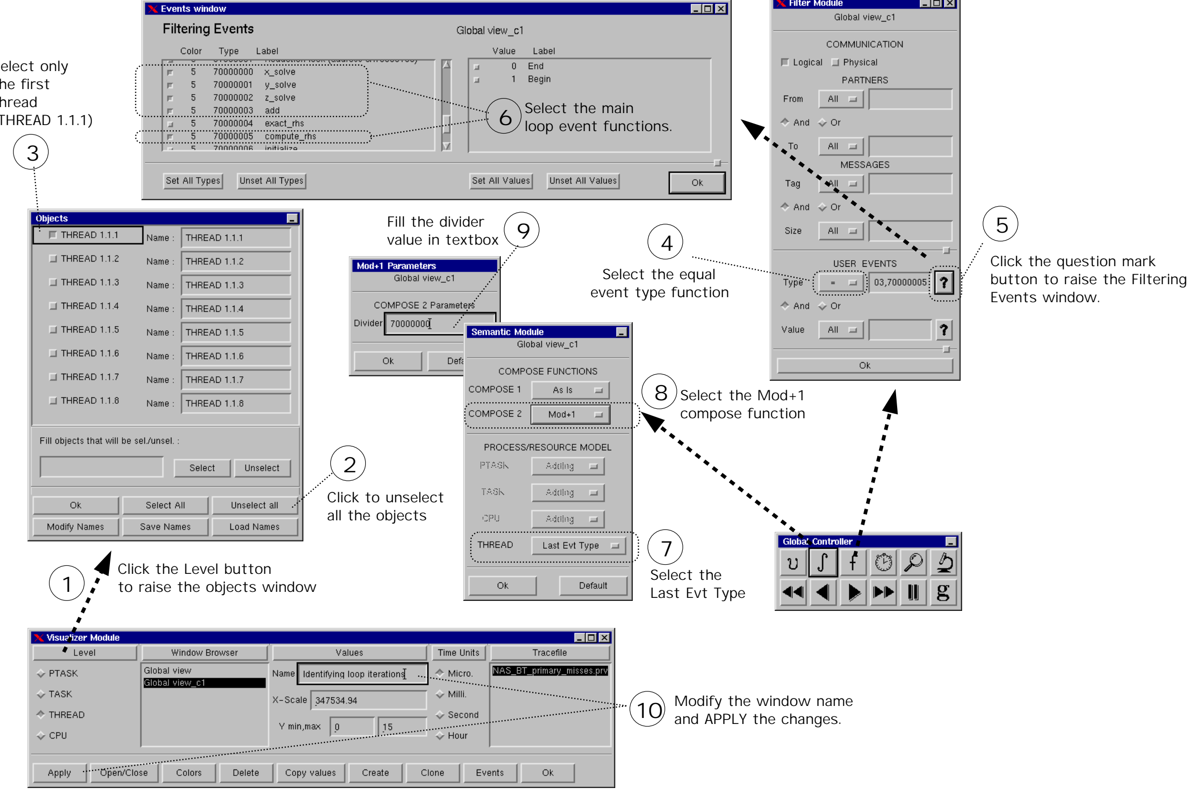


Figure 3.15: Identifying loop iterarions. OpenMP Instrumentation.

To solve it, first take as the current window the **Global view_c1** window. Raise the OBJECT WINDOW by clicking the LEVEL button in the Visualizer Module window (step **1** in figure 3.15) and unselect all thread objects except the THREAD 1.1.1 (step **2** and **3**). **Why ?** Because only master thread has marked these entries and exits.

Now, we have to filter all the events except the desired events, so raise the Filter Module, select the equal function in user event type menu (step **4**), raise the filtering events window by clicking the question mark button (step **5**) and select only the event types referring to the five main loop functions (**compute_rhs**, **x_solve**, **y_solve**, **z_solve**, **add**) like step **6**.

The next step is to select how information will be extracted. We have filtered only the desired user events but : **how will the information be extracted ?**

We are going to play with user the event types, note that each function have a different event type so we can obtain a different value for each function. Raise the Semantic Module and select at thread level the **Last Evt Type** function (step **7**). To obtain a small value select at COMPOSE 2 level the function **Mod+1** function (step **8**) with a divider value of 70000000 (step **9**). the table **??** shows how values will be converted.

Function name	Event Type	Mod+1 (divider: 70000000)
compute_rhs	70000005	6
x_solve	70000000	1
y_solve	70000001	2
z_solve	70000002	3
add	70000003	4

Table 3.7: Semantic Value returned by Last Evt Type + (Mod+1). OpenMP Instrumentation.

Finally, change the window name to **Identifying loop iterations** and apply the changes by clicking the APPLY button (step **10**). The window will be redrawn.

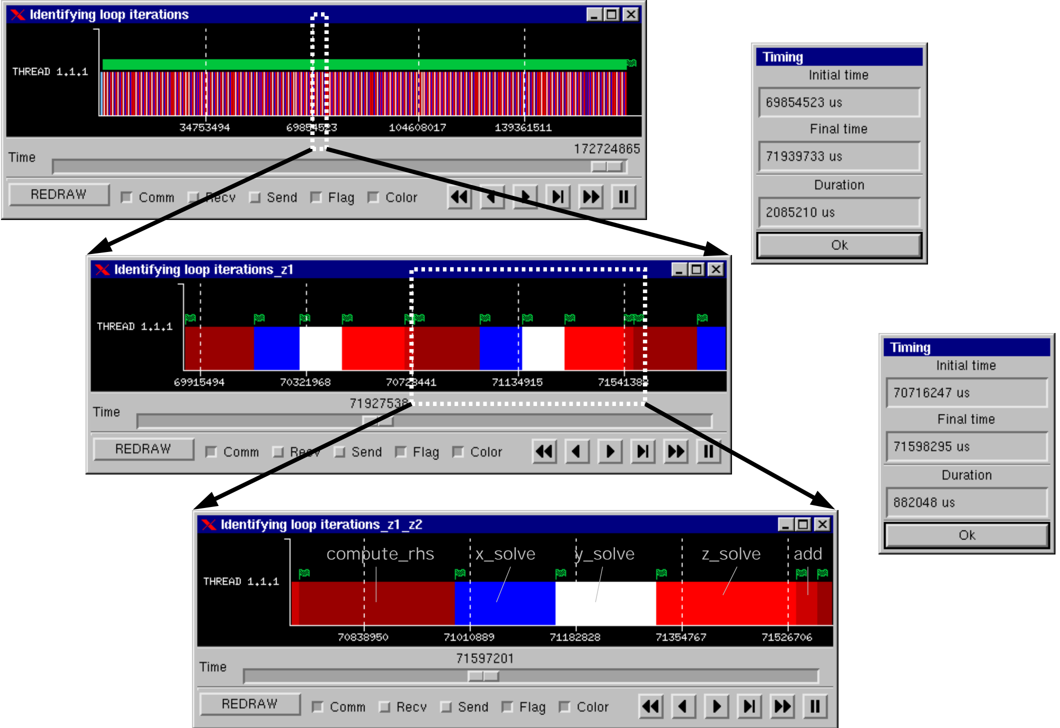


Figure 3.16: Identifying loop iterations window. OpenMP Instrumentation.

Each window will be painted with a different color (see figure 3.16). Make zooms like figure 3.16 until obtain only one iteration.

The window which contains only one iteration of the loop will be saved to load it in next sections. Therefore, change its name to **Loop iteration** and save it in **loop_iteration.cfg** file name.

3.6.2 Analyzing the five main functions.

Analyzing how many time has been within compute_rhs function.

The NAS BT application structure events let us to make an analysis to compute how much time the application has been within those functions. A very simple trick with the events let us to compute those values. Let's go to see how it can be computed.

First, select **Identifying loop iterations** window like the current. Our goal is to create a window where only the execution of **compute_rhs** function will be showed as running state, the rest of the functions have to be showed as idle state (value 0).

Remember that **compute_rhs** entries and exits have been coded by an event type where its value at the entry is 1 and its event value at the exit is 0. Thus, selecting a visualization that works with the event values will give us the desired visualization.

Raise the Semantic module, select the **As Is** function in COMPOSE 2 (step **1** in figure 3.17) and select the **Last Evt Val** function at thread level (step **2**).

Go to the Filter module and select only the **compute_rhs** event type, the rest have to be filtered (step **3**).

Finally, change its name to **Inside compute_rhs** and apply the changes (click the **Apply button**, step **4**). As a result we obtain a window like figure 3.18 where the master thread is in a

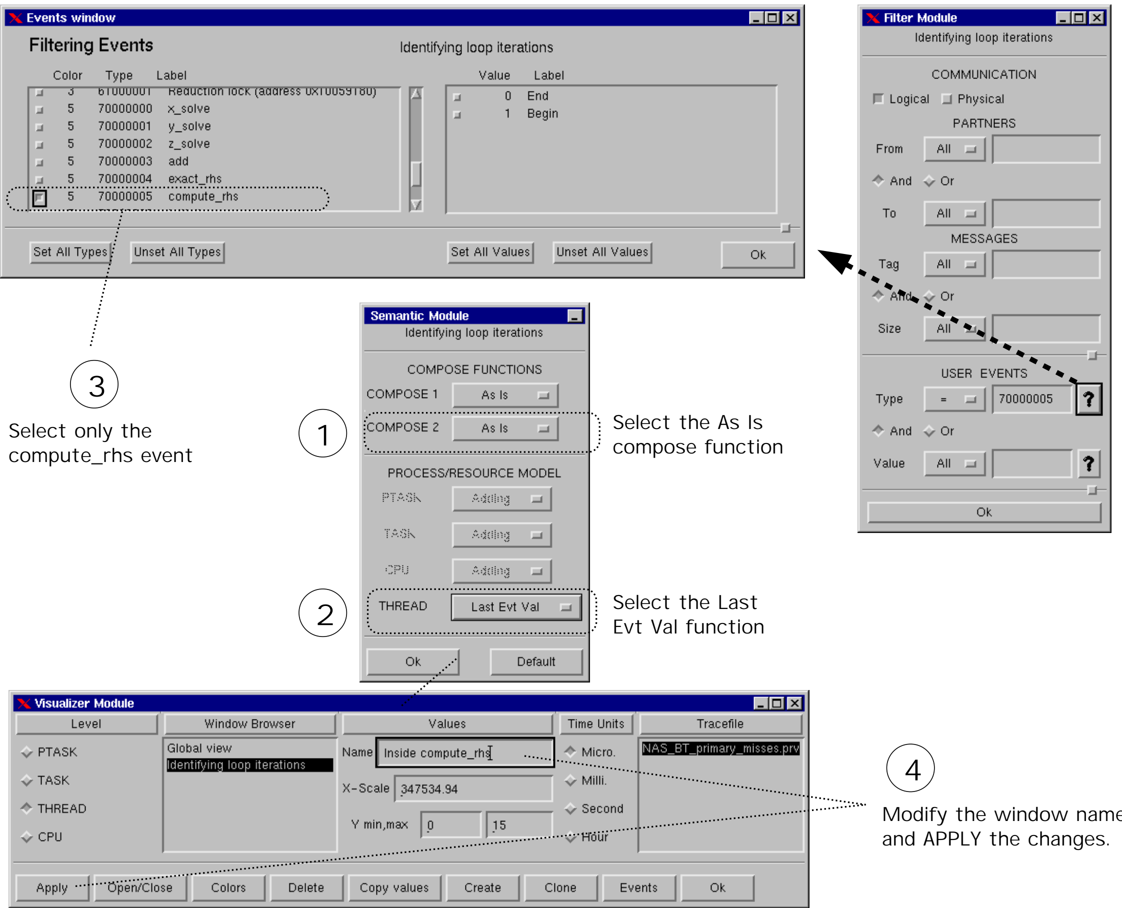


Figure 3.17: Selecting function compute_rhs. OpenMP Instrumentation.

working state when it is within the **compute_rhs** function, and in a non working state when he is out.

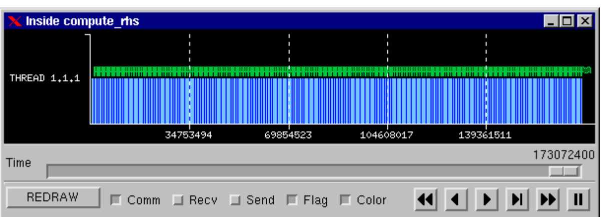


Figure 3.18: compute rhs function. OpenMP Instrumentation.

To compute how much time where the application has been executing this function, make an **All trace** analysis for this window and the **Average Semantic Val** function give this percentage, a value of 0.30. It means that the 30% of the execution time has been executing the compute_rhs function.

Furthermore, we could obtain the average duration of each execution (**Average Burst**), how many times has been executed (**# Burst**) and the standard desviation between execution times (**Stdev Burst**). Note, that the average time of each compute_rhs call is 255974.98 us (256 milliseconds), it has been executed 202 time and the standard desviation between execution times has been 10291.07 us (10 milliseconds).

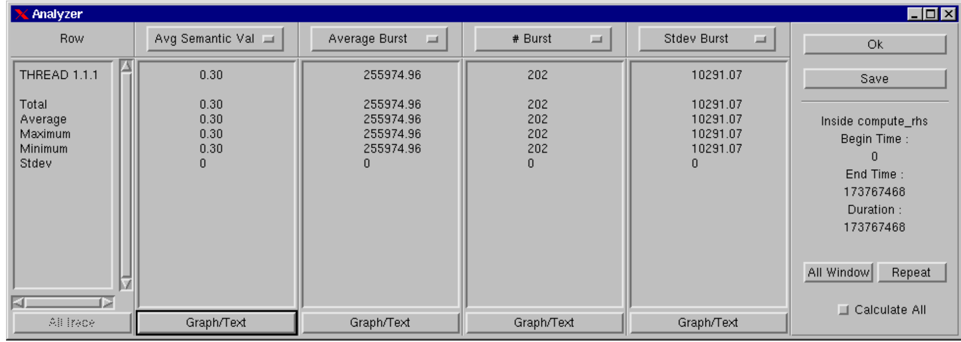


Figure 3.19: compute_rhs analysis. OpenMP Instrumentation.

The same process can be done for the other four functions, table 3.8 shows the resulting percentages :

Function	Percentage
compute_rhs_	0.30 (30 %)
x_solve_	0.19 (19 %)
y_solve_	0.19 (19 %)
z_solve_	0.27 (27 %)
add_	0.04 (4 %)

Table 3.8: Time within a function. OpenMP Instrumentation.

Note, that compute_rhs function and z_solve execution times are greater than others. Adding these percentatges we can conclude that these five functions are the 99 % of the execution time.

Analyzing compute_rhs parallel structure

The trace file contains information about NAS BT structure. Furthermore, it contains information about functions structure (parallel code within them have been marked by USER EVENTS), so we can make a view to show how functions execute its parallel code.

We are going to create a set of windows to show the **compute_rhs** structure (this function have different parallel regions, one parallel region and four parallel loops).

First, select the window called **Inside compute_rhs** as the current and zoom it until only one execution of **compute_rhs** will be displayed (like figure 3.20). You should try to fill a complete call within the window limits (when window has been created, rename it to **Inside one call of compute_rhs**).

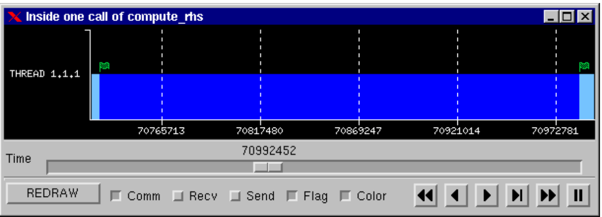


Figure 3.20: Only one call. OpenMP Instrumentation.

Now we are going to create a window where parallel functions will be displayed using a gradient color. In the paraver configuration files, gradient color names have been renamed to the parallel function names. Therefore, when we click in the next gradient visualization its function name will be displayed.

Since only master thread has the parallel function event type (see its definition in table 3.5 on page 29) we should create a window where only master thread will be displayed.

Take the window called **Inside one call of compute_rhs** as the current and clone it. Then, go to the filter module and filter all the event types except the parallel function event. At THREAD level in Semantic Module there should be select the **Last Evt Val** function. Raise the **Colors** window by clicking the COLOR button, and select the gradient visualization by clicking the *Gradient Mode* toggle button and change the Y-scale, Y min scale to 0 and Y max scale to 19 (the maximum parallel function event value is 19). Finally, change its name (to **Parallel functions inside compute_rhs**) and apply the changes by clicking the APPLY button. The window will be redrawn.

Figure 3.21 shows the resulting gradient visualization of parallel functions, clicking by the function the Textual module will give each function name.

Also, copy the window limits to the **Gloval view window** (press the COPY VALUES button and click the Global view window in the *Window browser*) to see the parallel execution within compute_rhs function (figure 3.21).

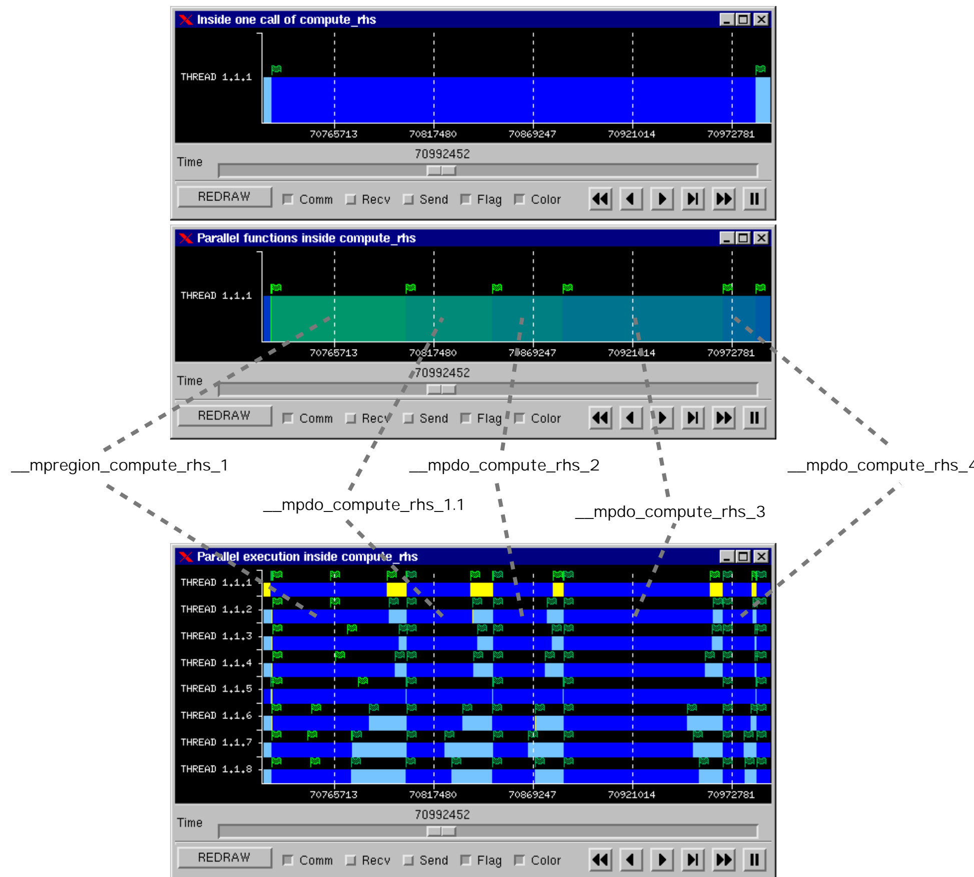


Figure 3.21: Only one call. OpenMP Instrumentation.

Synchronize different windows by copying its values is very useful in a way to compare different kind of displayed information from the same trace file.

3.7 Showing benchmark data cache misses.

The two provided trace files contain information about cache misses. As we explained in **What is the trace ? A brief description** section (see page 25) this information has been coded using user events (see table 3.6 on page 29).

In this section we are going to explain how this information could be extracted using the different paraver modules. We only are going to create a visualization profile of primary data cache misses (trace file *NAS_BT_primary_misses.prv*) but the same visualization could be done with secondary data cache misses (trace file *NAS_BT_secondary_misses.prv*), window configuration files has been supplied to obtain the same visualization.

3.7.1 Showing data cache misses profile

We are going to create a window which will display the primary data cache misses and we are going to explain how to identify the different regions with have bad cache behaviour.

You shouldn't have any window, so first we are going to load the **Global view** window saved in previous sections. Go to the CONFIGURATION menu option and **load** the configuration window that has been saved with the global trace file view (file *global_view.cfg*); the window will be created.

When this window has been created, clone it to obtain the **Global view_c1** window.

Creating data cache misses profile

Primary data cache misses have been coded in a user event type where their values are the number of cache misses done from previous read to current read (within previous interval); thus, to create the primary data cache misses visualization the semantic module has to work with user event values.

Raise the **Semantic Module** window and the **Filter window**, clicking onto the Semantic icon $\int$ and the Filter icon f on the *Global Controller window.*

Take the Semantic Module window, go to the THREAD pop up sub menu, and select the function called **Next Evt Val** (step **1** in figure 3.22). This function makes that the semantic module will work with event traces, and the semantic values passed to the upper levels will be their event values (in our selected function the value of the next event).

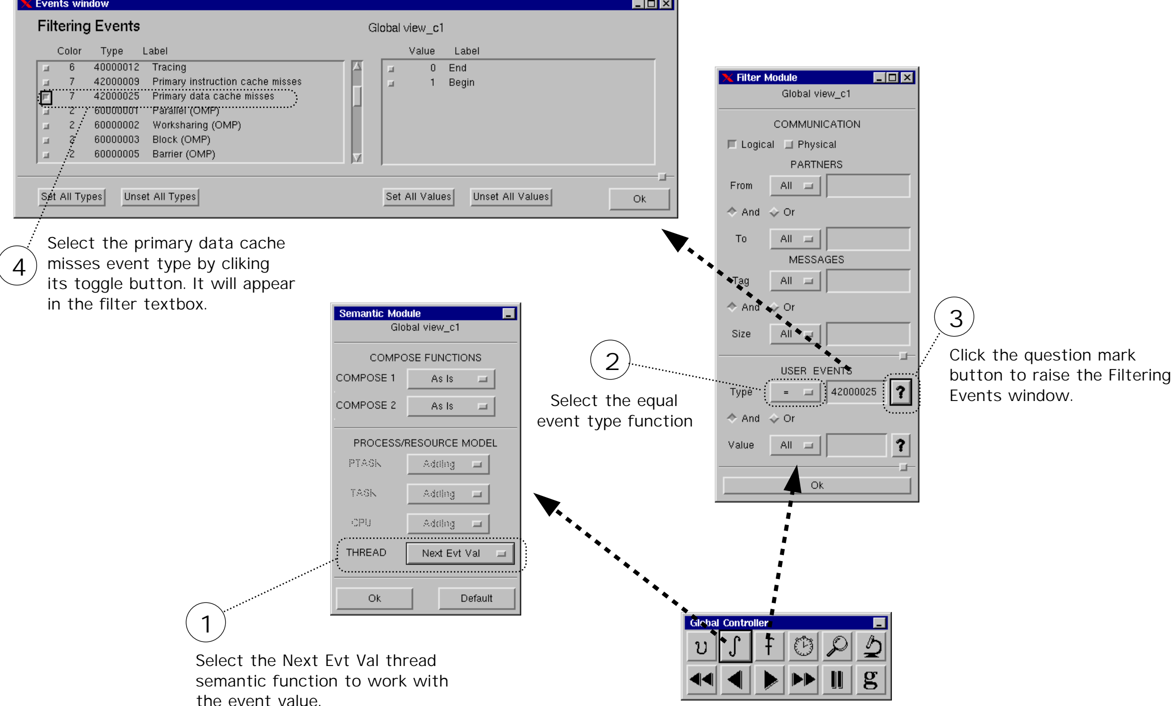


Figure 3.22: Showing data cache misses profile (I). OpenMP Instrumentation.

Why Next Evt Val ? We use the *Next Evt Val* because it returns the value of the next event, thus, it returns the number of primary data cache misses occurred in the interval that will be drawn (for more information see the SEMANTIC MODULE chapter on **Paraver Reference Manual**).

Also, we have to filter all the events except the *Primary data cache misses* (event type 42000025) because the trace file has more than one event type and we aren't interested in the rest event type values. To filter it, go to the USER EVENTS section in the Filter Module window, select the symbol "=" in the TYPE pop up sub menu (step **2**) and click the question mark button to raise the **Events window** (step **3**). In the Events window select the primary data cache misses (event type 42000025) by enabling its toggle button (step **4**). Through the *Semantic* and *Filter* modules, we have selected the semantic values that will be passed to the Representation Module. These semantic values are the event values for event type 42000025 (Primary data cache misses). These values could be a in a large range, so the color visualization won't work well.

We have to select a non-color visualization where we should select the correct Y scale (Y min should be the lowest value -value 0- and Y max should be the highest value -it has to be computed-). To compute the maximum we could use the **Max Semantic Val** function in the Analyzer Module. To compute it, click the **Analyzer button** (step **5** in figure 3.23) and select this function in a column of the Analyzer window (step **6**). Then, click the **All trace** button (step **7**) to compute it (figure 3.23).

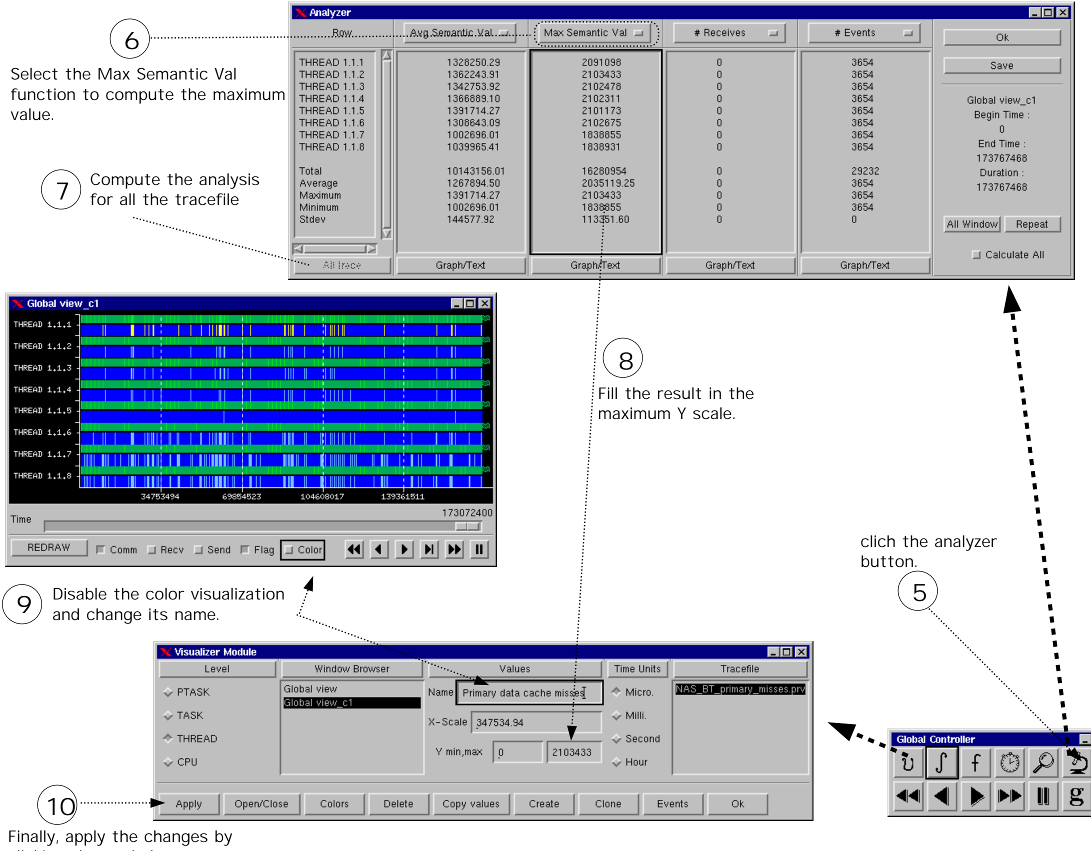


Figure 3.23: Showing data cache misses profile (II). OpenMP Instrumentation.

We have computed the maximum semantic value for primary data cache misses. Fill this value in the Y max text box in the Visualizer Module window (step **8**), and also, change the name of the window to **Primary data cache misses** (fill it in the Name text box in the Visualizer Module, step **9**).

Then disable the toggle Color button in the displaying window to obtain a non-color visualization (step **9**) and click the **Apply** button to apply the changes (step **10**). The window will be redrawn with the new parameters (figure 3.24). The resulting window shows the primary data cache misses profile for each thread where the Y-scale for each thread goes from 0 to the maximum value.

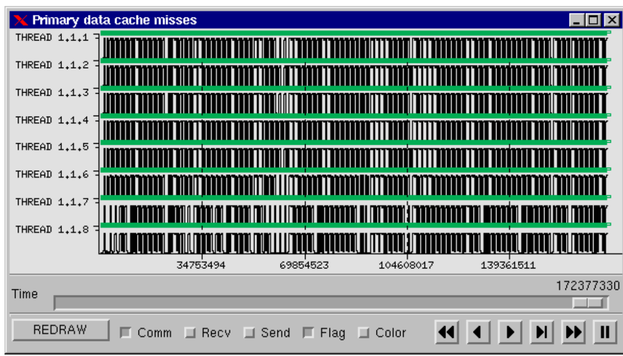


Figure 3.24: Data cache misses profile. OpenMP Instrumentation.

Make a zoom to see the details (figure 3.25). Note, that data cache misses are periodic, corresponding to the loop iterations. Data cache misses are showed for each thread, seeing them at PTASK level we could see the data cache misses done by all the application.

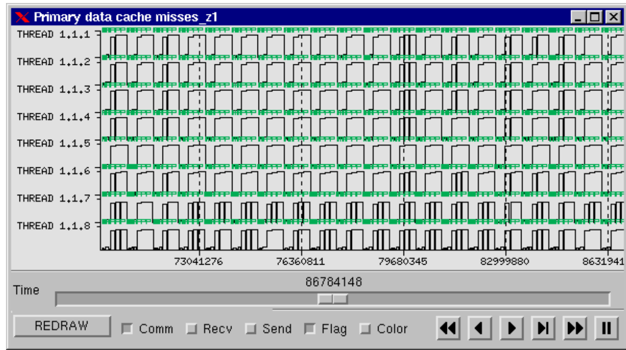


Figure 3.25: Data cache misses profile. OpenMP Instrumentation.

Another interesting view could be a visualization of the data cache misses done by a loop iteration. In previous sections, we created a window which contains one loop iteration (saved in file *loop_iteration.cfg*) so visualization can be done easily.

First, load the window configuration file *loop_iteration.cfg* which contains the loop iteration. To see the data cache misses in an iteration copy the window limits of the **Loop iteration** window to the zoomed window. The data cache misses profile zoom will be redrawn with the data cache misses done in an iteration (figure 3.26)

3.7.2 Showing data cache misses in function of time

The previous profile shows the number of primary data cache misses. In the previous section, intervals are painted with the number of cache misses occurred within. Another interesting profile could be a visualization of the primary data cache misses in function of time.

Long intervals could have a big number of data cache misses, if we take a smaller interval its value could be smaller. However, if we compute the data cache misses in function of time, the smaller interval could have a number of cache misses per time greater than the longer interval.

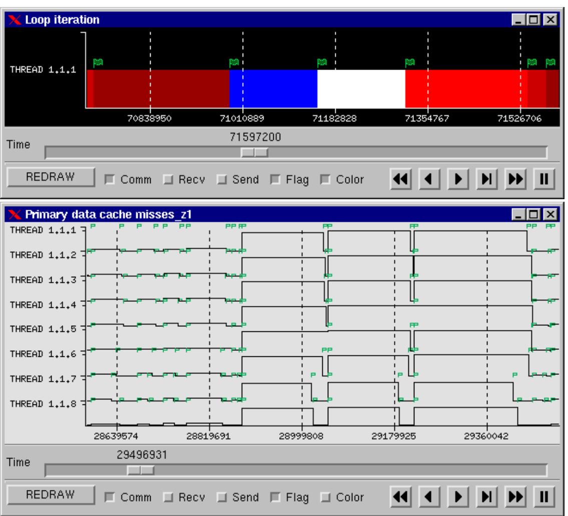


Figure 3.26: Data cache misses in an iteration. OpenMP Instrumentation.

This type of view will give to the user information about which intervals have the big number of data cache misses per time.

Creating data cache misses profile in function of time

Our goal is obtain a view which displays as values the number of primary data cache misses in function of time. There is a semantic function at thread level which will extract from the trace file these values, the **Avg Next Evt Val**.

Take the **Primary data cache misses** window as the current (select its name in *Window browser*) and clone it (press the CLONE button) to obtain the window called **Primary data cache misses_c1**.

Take the Semantic Module window, go to the THREAD pop up sub menu, and select the function called **Avg Next Evt Val** (step **1** in figure 3.27). This function makes that the semantic module will work with event traces, and the semantic values passed to the upper levels will be their event values in function of the duration.

Why Avg Next Evt Val ? We use the *Next Evt Val* because it returns the value of the next event so it returns the number of primary data cache misses occurred in the interval that will be drawn. Also, it will take as a duration of the interval the time between the current event and the next event (for more information see the SEMANTIC MODULE chapter on **Paraver Reference Manual**).

We have to compute the correct Y scale because the semantic function has changed and the scale could change (Y min should be the lowest value -value 0- and Y max should be the highest value). To compute the maximum we could use the Analyzer Module with function **Max Semantic Val**. To compute it, click the **Analyzer button** (step **2** in figure 3.27) and select this function in a column of the Analyzer window after click onto the Analyzer icon (step **3**). Then, click the **All trace** button (step **4**) and the result will be displayed (figure 3.27) after computing the analysis. Fill this value in the Y max text box in the Visualizer Module window (step **5**), and also, change the name of the window to **Primary data cache misses** (fill it in the Name text box in the Visualizer Module, step **6**).

Then disable the toggle Color button in the displaying window to obtain a non-color visualization (step **6**) and click the **Apply** button to apply the changes (step **7**). The window will be redrawn with the new parameters (figure 3.28) and we can see the values of the primary data cache misses profile in function of time for each thread.

Make a zoom to see the details (figure 3.29). Note that profile is different from profile when

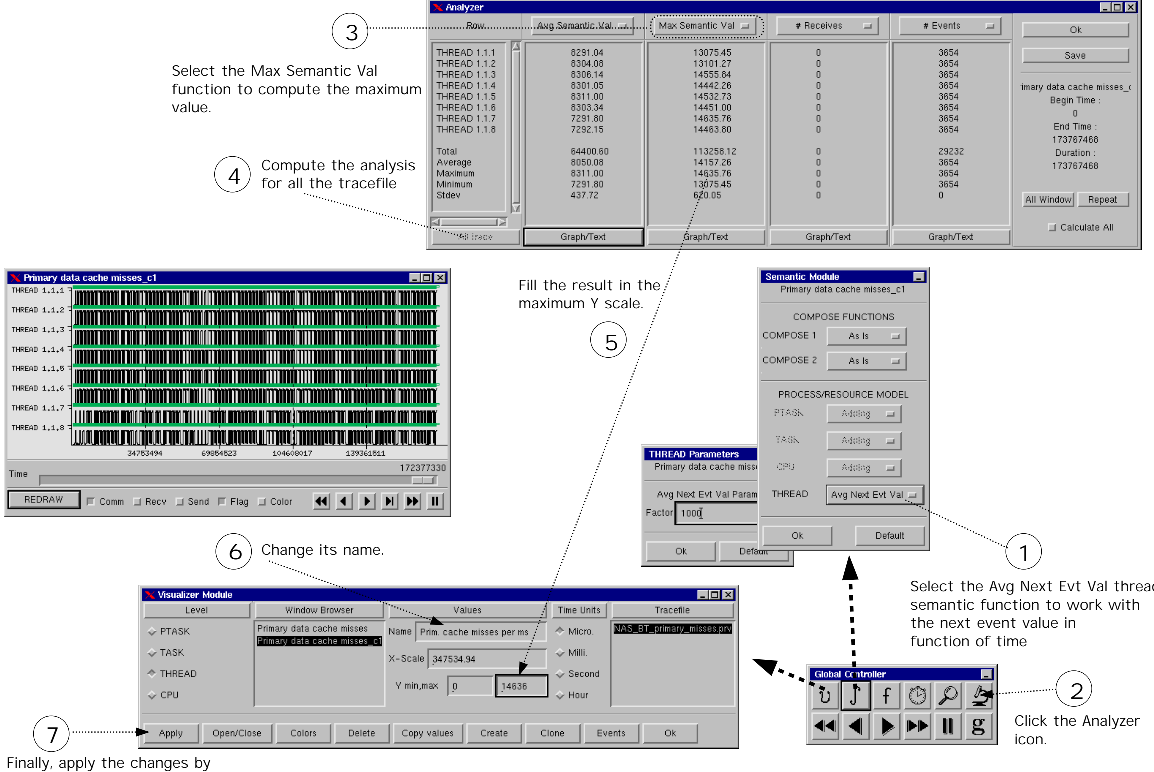


Figure 3.27: Showing data cache misses profile in function of time. OpenMP Instrumentation.

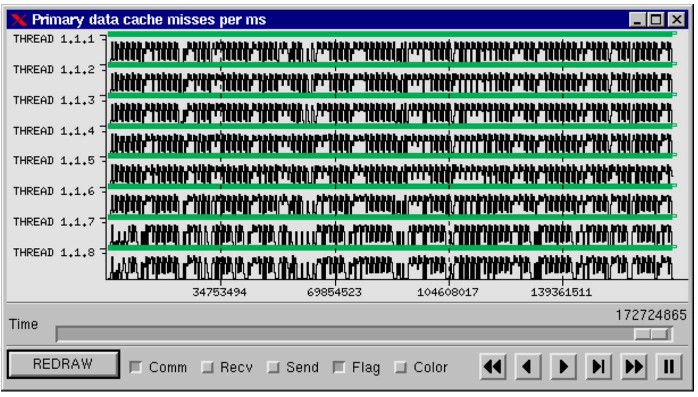


Figure 3.28: Data cache misses profile in function of time. OpenMP Instrumentation.

only data cache misses values are displayed. Regions where in a small time make many primary data cache misses are showed with a greater value because they are showed in function of time. Remember, that FACTOR selected was 1000 so we are computing the number of primary data cache misses per millisecond, if we select a value of 1 we compute the data cache misses per microsecond, ...

As previous section, an interesting view could be show the data cache misses done by an iteration. To show it, copy the window limits of the **Loop iteration** window to the previous zoomed window. As result we obtain figure 3.30.

Compare these window with the primary data cache misses visualization in an iteration. Regions were there were many cache misses are visualized when working in function of time, but new

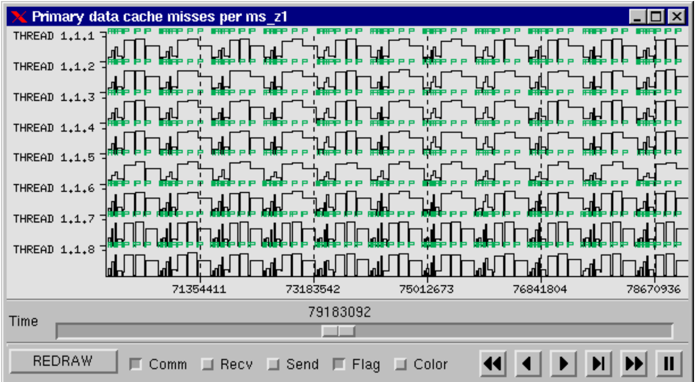


Figure 3.29: Data cache misses profile in function of time. OpenMP Instrumentation.

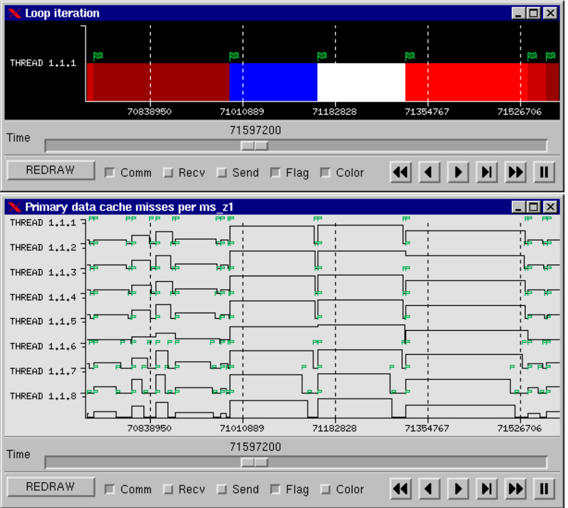


Figure 3.30: Data cache misses in function of time in an iteration. OpenMP Instrumentation.

regions have been appeared. Those regions are sections of code where there were a lot of cache misses in a little interval.

3.8 Window configuration files supplied for this chapter.

We have been supplied some window configuration files to load predefined window which shows different aspects of the trace file.

These files can be found in directory *tutorial_traces/omp_nas_bt/cfg_examples* in tutorial traces package.

- *primary_data_cache_misses_PTASK.cfg* for trace file **NAS_BT_primary_misses.prv**. It shows the primary data cache misses profile done by all threads (at PTASK level). This window configuration file will create four windows : a general view, a zoom of the general view, an iteration and primary data cache misses profile in an iteration.

- *primary_data_cache_misses_per_ms_PTASK.cfg* for trace file **NAS_BT_primary_misses.prv**. It shows the primary data cache misses per milliseconds profile done by all threads (at PTASK level). This window configuration file will create four windows : a general view, a

zoom of the general view, an iteration and primary data cache misses per milliseconds profile in an iteration.

- *secondary_data_cache_misses_profile.cfg* for trace file **NAS_BT_secondary_misses.prv**. It shows the secondary data cache misses profile. This window configuration file will create four windows : a general view, a zoom of the general view, an iteration view and secondary data cache misses profile in an iteration.
- *secondary_data_cache_misses_per_ms_profile.cfg* for trace file **NAS_BT_secondary_misses.prv**. It shows the secondary data cache misses per millisecond profile. This window configuration file will create four windows : a general view, a zoom of the general view, an iteration view and secondary data cache misses per millisecond profile in an iteration.
- *secondary_data_cache_misses_profile.cfg* for trace file **NAS_BT_secondary_misses.prv**. It shows the secondary data cache misses profile done by all the threads (at PTASK level). This window configuration file will create four windows : a general view, a zoom of the general view, an iteration view and secondary data cache misses profile in an iteration.
- *secondary_data_cache_misses_per_ms_profile.cfg* for trace file **NAS_BT_secondary_misses.prv**. It shows the secondary data cache misses per millisecond profile done by all the threads (at PTASK level). This window configuration file will create four windows : a general view, a zoom of the general view, an iteration view and secondary data cache misses per millisecond profile in an iteration.

Chapter 4

Hardware counters profile. NAS BT Benchmark

4.1 What is the trace ? A brief Description.

Our third example is composed by a set of trace files. Each trace file is the output file of a tool called infoPerfex which extracts the hardware counters over a Silicon Graphics Origin 2000 machine during the execution of the user application.

These trace files contain the values extracted from the hardware counters during the execution of the sequential application of the NAS BT. We have a set of trace files because not all the hardware counters can be read at the same time, and we did some executions reading the different hardware counters to obtain some examples.

During the benchmark execution, *infoPerfex*[1] reads the hardware counters more or less each 100 milliseconds (ms.) and codes them like user events, where the event type is the number of a hardware counter and the event value is its value in each reading. Thus, each trace file contains each 100 ms some user events representing the read to the hardware counters.

There are events common to all the trace files corresponding to hardware counters that ever can be read, but usually they don't appear because its read values were 0 (the paraver configuration file for each trace file has defined their labels).

4.1.1 Defined USER EVENTS

Hardware Counter events.

This example is composed by seven trace files where different hardware counters has been traced. Each hardware counter has been coded into a different event type where each read contains the value from last read until now. For example, each user event value of the event type which encodes the primary data cache misses is the number of cache misses occurred from the last read until this read.

Table 4.1 shows for each trace file which specific events includes. Some of these trace files that refers to instruction counters (trace files like *floating_point_instructions.prv*) have an issued or a graduated value. Although they are counting the same type of instructions there is a big difference between them :

- the **issued** value contains the number of instructions taken from the input queue and assigned to a functional unit for processing. Note, that some instructions can be issued more than once before they are completed, and some can be issued and then discarded (speculative execution). As a result, issued instructions reflect the amount of the work the CPU does.

[1]information about infoPerfex tool can be found at URL http://www.cepba.upc.es/tools/paraver/paraver.htm

Tracefile name	Event Type	Label
floating_point_instructions.prv	17	Graduated Instructions.
	21	Graduated floating instructions.
primary_cache_misses.prv	9	Primary instruction cache misses.
	25	Primary data cache misses.
secondary_cache_misses.prv	10	Secondary instruction cache misses.
	26	Secondary data cache misses.
tlb_misses.prv	13	External invalidations.
	23	TLB misses.
load_instructions.prv	2	Issued loads.
	18	Graduated loads
store_instructions.prv	3	Issued stores.
	19	Graduated stores
branch_instructions.prv	6	Decoded branches.
	24	Mispredicted branches.

Table 4.1: Specific Event Types. Hardware counters profile.

- the **graduated** instructions reflect the effective work towards the completion of the algorithm.

NAS BT application structure events.

Some user events have been used to mark the entry and exit of the main NAS BT functions. These type of user events has been traced manually.

The *NAS BT* benchmark has a main loop that calls five functions. The five functions are repeatedly executed. The main NAS BT loop looks like :

```
DO I=1, N
    compute_rhs
    x_solve
    y_solve
    z_solve
    add
END DO
```

Table 4.2 shows these event types and their labels. Note that the five main loop functions has been marked with different event types where their values mark the entry and exit to the function. Also, initialization routines have been marked (*initialize* and *exact_rhs*). Also, the beginning of an iteration and the ending has been marked with an event type.

Event Type	Label	Values
70000	compute_rhs	1 Begin
70001	x_solve	0 End
70002	y_solve	
70003	z_solve	
70004	add	
70005	exact_rhs	
70006	initialize	
80000	Loop iteration (adi.f)	

Table 4.2: Main loop function event types. Hardware counters profile.

We will see this structure in the analysis of our instrumentation example. The NAS BT benchmark could be executed with a different amount of data and iterations. For our working

example we used a class known as "class A", which do around 200 iterations to the loop which braces the five functions.

```
Class          =          A
Size           =   64x 64x 64
Iterations     =        200
```

On this example, we wouldn't make an analysis for all these trace files, we only want to show how paraver could be used to analyze the performance of an application.

First, launch Paraver and load the trace file called **floating_point_instructions.prv** that can be found in directory *tutorial_traces/hwc_nas_bt* in tutorial traces package[2].

When the trace file has been loaded, paraver asks for load its paraver configuration file (floating_point_instructions.pcf) which contains information about the user event labels explained in the previous section. Load it by clicking the LOAD button.

4.2 Visualization of Graduated floating point instructions.

Our first hardware counters visualization will be the graduated floating point instructions profile. Remember that traces used in this chapter are composed only by USER EVENTS so we have to work with semantic functions that work with user event traces.

Press the visualizer button $\boxed{v}$ in the *Global Controler* window and when the Visualizer Module window appears, press the **Create** button. This will create a displaying window named **win_1** where there is painted only the axis. Before, pressing the Play button enable the **Flag** toggle button in the displaying window to display the flags.

Then, press the **Play** button on the displaying window to obtain the visualization. This occurs because the trace file doesn't have state records and by default, paraver works with the useful view which works with working and non working states. Therefore, we have to work with event records.

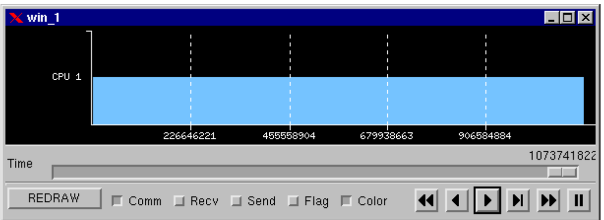


Figure 4.1: Working with states in infoPerfex traces. Hardware counters profile.

Now, we are going to select some parameters and properties of the displaying window to obtain a display visualization where we will see the values that has been taken for the **Graduated Floating Instructions** painted as a time line.

Raise the **Semantic Module** window and the **Filter window**, clicking onto the Semantic icon $\boxed{\int}$ and the Filter icon $\boxed{f}$ on the *Global Controler window.*

Take the Semantic Module window, go to the THREAD pop up sub menu, and select the function called **Next Evt Val** (step **1** in figure 4.2). This function makes that the semantic module will work with event traces, and the semantic values passed to the Representation Module will be their event values.

Why Next Evt Val ? We use the *Next Evt Val* because it returns the value of the next event so it returns the number of floating point instructions occurred in the interval that will be drawn (for more information see the SEMANTIC MODULE chapter on **Paraver Reference Manual**).

[2]traces are available in *Documentation tool* section at URL http://www.cepba.upc.es/tools/paraver/paraver.htm

Also, we have to filter all the event except the *Graduated Floating Point Instructions* (event type 21) because the trace file has more than one event type and we aren't interested in the other event type values. To filter it, go to the USER EVENTS section in the Filter Module window, select the symbol "=" in the TYPE pop up sub menu and click the question mark button to raise the **Events window** (step **3**). In the Events window select the graduated floating point instructions (event type 21) by enabling its toggle button (step **4**).

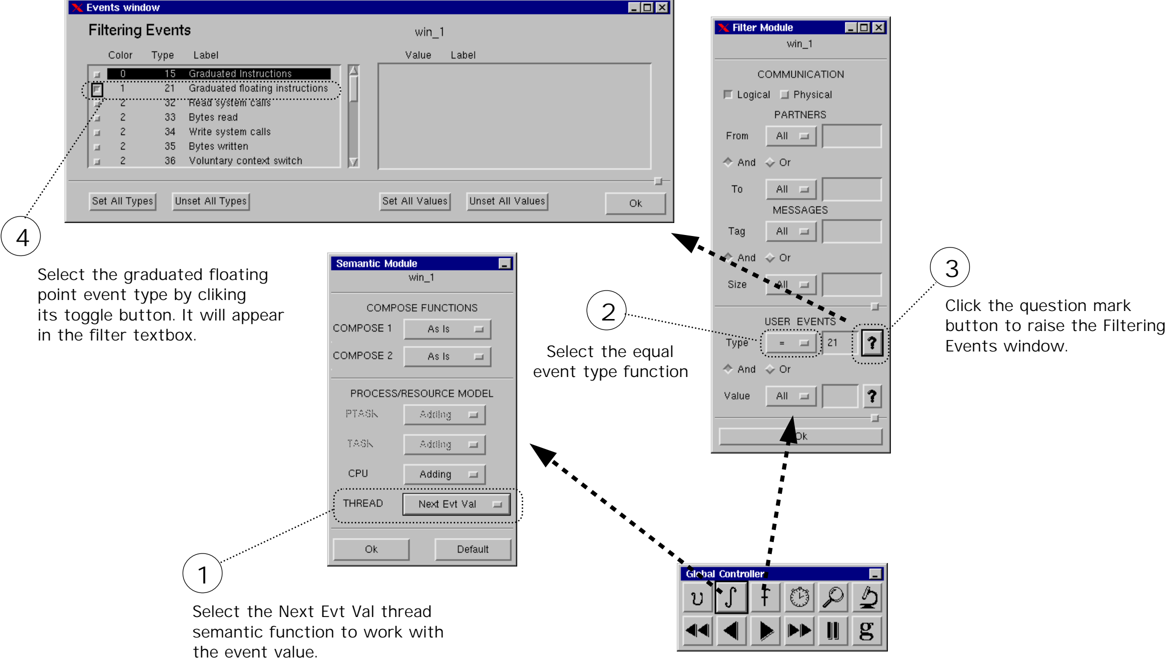


Figure 4.2: Selecting the "floating instructions" event (I). Hardware counters profile.

Through the *Semantic* and *Filter* modules, we have selected the semantic values that will be passed to the Representation Module. These semantic values are the event values for event type 21 (Graduated floating point instructions). These values could be a in a large range, so the color visualization won't work well.

We have to select a non-color visualization where we should select the correct Y scale (Y min should be the lowest value -value 0- and Y max should be the highest value). To compute the maximum we could use the Analyzer Module with function **Max Semantic Val**. To compute it, select this function in a column of the Analyzer window after click onto the Analyzer icon (step **5** in figure 4.3). Then, click the **All trace** button (step **6**) and the result will be displayed (figure 4.3) after computing the analysis.

We have computed the maximum semantic value for graduated floating point instructions. Fill this value in the Y max text box in the Visualizer Module window (step **7**), and also, change the name of the window to **Graduated floating instructions** (fill it in the Name text box in the Visualizer Module) to difference this window through the windows that will be created.

Then disable the toggle Color button in the displaying window to obtain a non-color visualization (step **8**) and click the **Apply** button to apply the changes (step **9**). The window will be redrawn with the new parameters (figure 4.4) and we can see the values of the floating point instructions hardware counter as a time line[3].

Figure 4.4 shows the profile of the counter *graduated floating point instructions*. Through this visualization, we could detect when performance goes up or down. To see the details you can

[3]To capture the windows we have changed the background and foreground (using the option OPTIONS/SYSTEM COLORS). If you work normally with Paraver without changing any color, foreground will be the white color and background will be the black color.

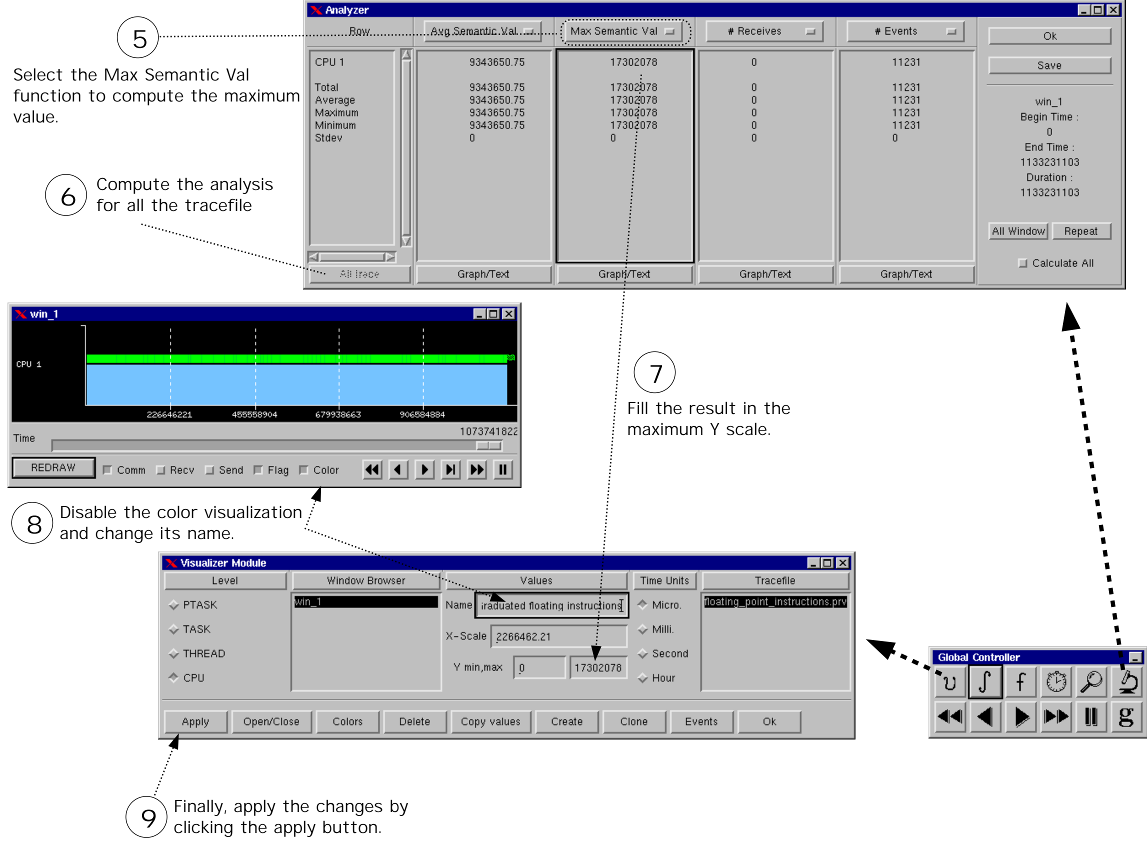


Figure 4.3: Selecting the "floating instructions" event (II). Hardware counters profile.

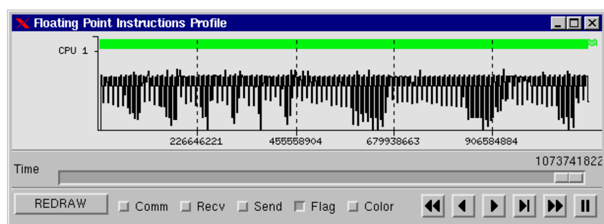


Figure 4.4: Graduated floating point resulting window. Hardware counters profile.

make a zoom like figure 4.5.

The floating point instruction profile is periodic because NAS BT is composed by a loop which braces five functions that are executed many times.

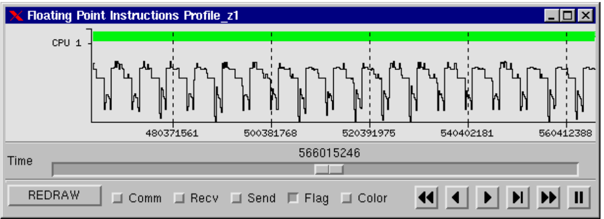


Figure 4.5: Graduated floating instr. zoomed window. Hardware counters profile.

We could use the **Analyzer** module to compute the MFlops of the application execution. To compute it, make an analysis for all the trace file computing the **Avg Semantic Val** function. In the trace file there is the number of floating point instructions that has been executed each more or less 100 milliseconds, the **Avg Semantic Val** function will give *the average number of floating point instructions each 100 milliseconds* and to obtain the MFlops we have to multiply this number by 10. In our example, the **Avg Semantic Val** is 9343650.75 flops per 100 ms (see figure 4.3) and multiplying it by 10 we obtain **93436507.5 flops per second** (more or less **93.5 MFlops**).

4.3 Visualization of data cache misses.

We could analyze the data misses that our application execution has been done. We are going to make three windows, one for primary data cache misses, other for secondary data cache misses and finally, the TLB misses that the application has been done.

4.3.1 Primary data cache misses.

To obtain a displaying window with a profile of the primary data cache misses we could do the same as we did to obtain the graduated floating instructions profile. First, we have to load the trace file called **primary_cache_misses.prv** which contains the hardware counter which provides this information. This hardware counter has been coded in the event type 25 (see Table 4.1).

First, create a new window, enable its toggle button and play it (like we have done with floating point instructions).

Go to the **Semantic Module window** and in the THREAD pop up sub menu, select the function called Next Evt Val to work with event values. Go to the **Filter Module window** and filter all the events except the event type 25 (in the **Type** pop up menu, select the "=" symbol and select the event type 25).

Remember, that you have to select the correct Y scale which includes all the event values of the trace file. To find this value, compute the **Max Semantic Val** for all the trace file (*All trace* button) and fill this value in the Y max scale. Also, you should change the window name to difference the windows (for example change to Primary data cache misses).

Finally, disable the color mode and apply the changes (click onto the APPLY button) to show our primary data misses window.

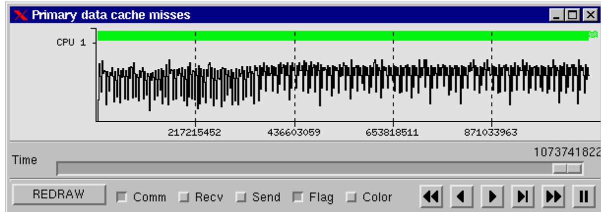


Figure 4.6: Primary data misses window. Hardware counters profile.

Make a zoom to see the details, note how the primary data cache misses are periodic too.

4.3.2 Secondary data cache misses.

The process to obtain a displaying window with the secondary data misses is like the process to obtain the primary data misses. First, you have to load the trace file called **secondary_cache_misses.prv** which contains the secondary data cache misses counter. This counter has been coded in event type 26 (see 4.1). By default, the new loaded trace file is selected as the current one.

The same steps than visualization of primary data cache misses should be done to obtain the window but here we have to filter all the events except the event type 26 which contains the desired

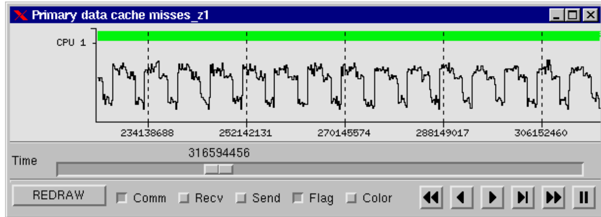


Figure 4.7: Primary data misses zoomed window. Hardware counters profile.

information. Also, a new Y scale should be computed, in this trace file the maximum value for the secondary data cache misses is 208138; note, that this value is smaller than primary data misses.

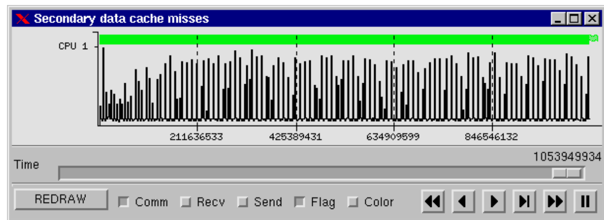


Figure 4.8: Secondary data misses window. Hardware counters profile.

As for the primary data cache misses, make a zoom to see the details. Note that the behavior is the same the application has memory problems in certain points each iteration of the external loop.

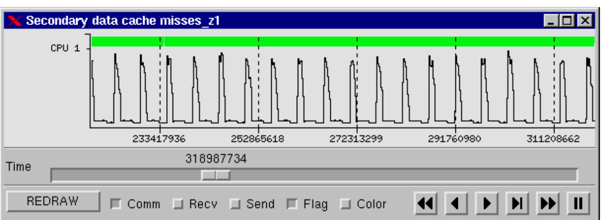


Figure 4.9: Secondary data misses zoomed window. Hardware counters profile.

4.3.3 TLB data misses.

Finally, we could obtain a displaying window to see the TLB misses along the execution. You have to load the trace file called **tlb_misses.prv** where the event type 23 contains these values.

Do the same process like but filtering all the events except the event type 23. The correct Y scale in when we are displaying this value that we have computed should be the value 270635.

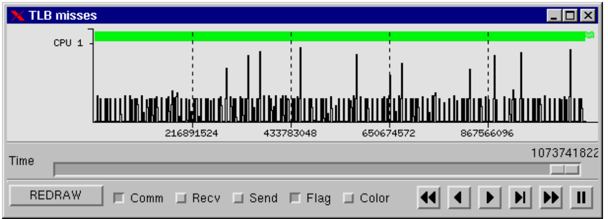


Figure 4.10: TLB data misses window. Hardware counters profile.

The zoom will give us a better visualization of TLB misses.

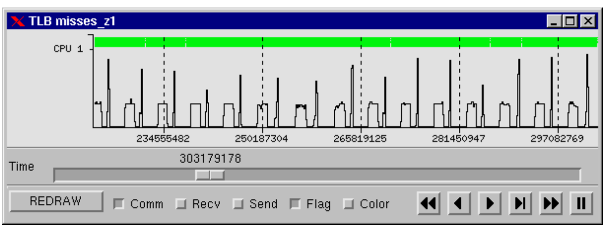


Figure 4.11: TLB data misses zoomed window. Hardware counters profile.

4.4 Instruction set used by the application

Through the hardware counters, we could analyze how many instructions the application has been executing and the distribution of each type. We could extract information about the number of loads, stores, branches and floating point instructions and detect what regions could have problems, and where the performance goes down. These values could give us an information about what it is happening during the execution and we could relate it with the values showed on the previous section, for example we could relate with data misses during the application execution.

The process to obtain these displaying windows is the same than in the previous sections. You have to load the trace file which has the desired event type to study (see table 4.1), filter it, and select the desired limits to work with those event values.

Also, the *Analyzer* module lets us to compute things like how many instructions are executed each time and how many of each type.

For example, we could obtain a window like figure 4.12 that shows the graduated load instructions (event type 18) during the execution of the trace file.

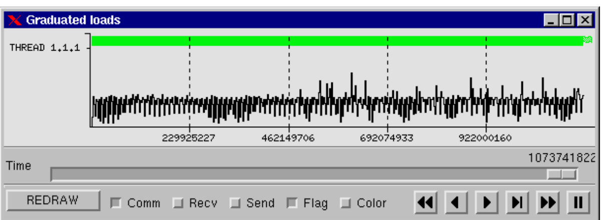


Figure 4.12: Graduated Loads. Hardware counters profile.

The average number of load instructions that the application has executed each second can be computed with the Analyzer Module. We only have to make an analysis for all the trace file and compute the **Avg Semantic Val** function. This value will be the average number of load instructions executed each 100 ms, to obtain the average per seconds we only have to multiply this value per 10.

The same analysis could be done for the graduated instructions, the graduated stores, the graduated floating point instructions (see section 4.2) and decoded branches. Those results are showed in table 4.3.

The table shows average number of instructions executed each second (Graduated instructions) and the average number of instructions for the floating point, loads, stores and branches instruction types. Adding these four types results an average of 219,937,317.1 instructions per second. If we substract this value from the average number of instructions per second results a **9,754,764.3 instructions which are the average of aritmethical integer**.

Instruction Type	Instructions per second
Graduated instr.	229,692,081.4 Instr. per second.
Graduated floating instr.	97,436,507.5 Instr. per second.
Graduated loads	73,019,551.3 Instr. per second.
Graduated stores	47,750,960.2 Instr. per second.
Decoded branches	1,730,298.1 Instr. per second.

Table 4.3: Executed instructions. NAS BT BenchMark

4.5 Window configuration files supplied for this chapter.

We have been supplied some window configuration files to load predefined window which shows different aspects of the trace file.

These files can be found in directory *tutorial_traces/hwc_nas_bt/cfg_examples* in tutorial traces package.

- *floating_point_instructions_in_an_iteration.cfg* for trace file **floating_point_instructions.prv**. It shows the floating point profile in an iteration where each of the five functions is painted using a different color.
- *primary_cache_misses_in_an_iteration.cfg* for trace file **primary_cache_misses.prv**. It shows the primary data cache misses profile in an iteration where each of the five functions is painted using a different color.
- *secondary_cache_misses_in_an_iteration.cfg* for trace file **secondary_cache_misses.prv**. It shows the secondary data cache misses profile in an iteration where each of the five functions is painted using a different color.
- *tlb_misses_in_an_iteration.cfg* for trace file **tlb_misses.prv**. It shows the tlb misses profile in an iteration where each of the five functions is painted using a different color.

Chapter 5

Multiprogrammed Executions. Visualization and Analysis

5.1 What the trace is ? A brief Description.

The last example is a trace file containing the physical processor allocation done by the operating system during the execution of a workload composed by different parallel applications.

These traces have been collected by a tool called **SCPUs**[1] which collects information about the processor allocation by sampling at a given time step. This tool runs simultaneously with the workload collecting instantaneous machine status and generating a textual trace.

The workload has been executed in a Silicon Graphics Origin 2000 machine with 64 processors running IRIX 6.5 in exclusive mode, therefore, the trace file shows the physical processor allocation of IRIX operating system. The trace file generated by **SCPUs** tool contains the mapping of the application threads of the workload execution to physical processors plus the process migrations between processors (represented as communications). **SCPUs** tool maps the real execution traces in a Paraver trace file to visualize and analyze the workload execution.

The **SCPUs** tool samples the system processor allocation at a given time step specified by the user. To obtain our trace file we sampled more or less each 100 milliseconds so we are supposing that between this interval the application has been running in the same processor (a more exhaustive sampling could be done using the **SCPUs** tool). In each sample the **SCPUs** tool saves which threads are in each physical processor, and when the workload finishes we have the traces of where each application thread has been executed during the workload.

5.1.1 Workload description

The workload consists of six parallel applications (using old MP and new OpenMP directives) requesting a different number of processors. Table 5.1 shows the number of processors requested by each application.

Note that the number of requested processors by all the workload goes to 96, but the machine only has 64 physical processors, there are more requested processors than available in the machine so the operating system should distribute the resources between the applications. The trace file shows the processor allocation that has been done by the IRIX operating system, but using these kind of trace files different processor allocation policies can be studied.

5.1.2 Defined STATES

The workload is composed by different applications. These applications have been mapped to states and all the threads within the application are painted using the same color. Table 5.1 shows

[1]information about SCPUs tool can be found at URL http://www.cepba.upc.es/tools/paraver/paraver.htm

the mapping between the states and each application in the workload.

State Value	Application	Requested processors
0	Idle (processor isn't working).	
1	BT Application.	16 processors
2	ltomcatv Application.	16 processors
3	Swim Application.	16 processors
4	Turb3D Application.	24 processors
5	Hydro2D Application.	16 processors
6	Su2cor Application.	8 processors

Table 5.1: Mapping between application and states. Multiprogrammed executions

Mapping applications to state values will let to paint, using different colors, which application is running in each physical processor.

5.1.3 Defined COMMUNICATIONS

The thread migrations between the different processors has been coded as communications because a thread migration could be seen as a message from *processor X* to *processor Y*. The communication tag is the number of application which makes the thread migration. No migration is stored if thread is scheduled on the same processor than in its last run and only a color change will be displayed if a different application begins to run in that processor.

The tag value lets to filter the communication made by a specific application during the workload and focus our study on one application filtering the rest (see Selecting one application section on page 67).

5.2 How does it look : Visualization.

First, launch Paraver and load the trace file called **scpus.prv** that can be found in directory *tutorial_traces/scpus_workload* in tutorial traces package[2].

When trace file has been loaded, **paraver** asks for load its paraver configuration file (*scpus.pcf*). Load it by clicking the LOAD button.

This file defines the mapping between states and applications, so the textual display tool will give the application name for a specified color. Also, this file changes the default color palette used by Paraver to differentiate the applications because the default palette has similar RGB color for states 3, 4, 5 and 6 (the default configuration use them as blocking states so their are painted with a similar red color).

The mapping between state values and applications could be seen in the COLORS WINDOW (see figure 5.1) or clicking onto the trace file with the TEXT MODE enabled.

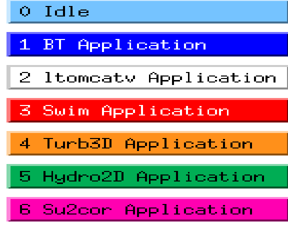


Figure 5.1: Mapping between colors and applications. Multiprogrammed executions

[2]traces are available in *Documentation tool* section at URL http://www.cepba.upc.es/tools/paraver/paraver.htm

To create the first view of the trace file, raise the *Visualizer Module* window by clicking onto the visualizer module icon $\boxed{v}$ in the *Global Controller* window and press the CREATE button. This will create a window with 64 rows at processor (CPU) level (one for each physical processor), note that this window is bigger than the previous examples, so the number of windows that could be created depends on the X Server memory.

Each row shows a physical processor, where the X-axis show which applications have been running in each processor along the time.

The CREATE button has created a window were only the axis are painted. Just by pressing the **Play** button, the trace file will be drawn in the displaying window (figure 5.2).

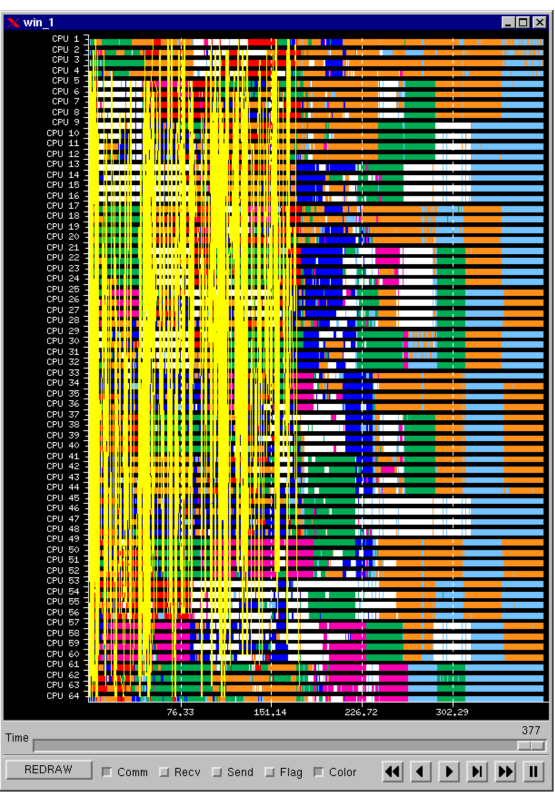


Figure 5.2: Processor allocation trace file. Multiprogrammed executions

The displaying window shows the communications, they represent the process migrations between processors and in each row, using a different color, shows the applications running each moment. Light blue color means that the processor is idle, at the end of the workload some processors begin to stay in an idle state because some applications have finished and the rest did not requested enough to use all the processors.

Note that during the workload execution there is a lot of process migrations and when applications begin to finish the operating system continues ordering migrations between processors. However there are physical processors idle (there are less processors requested than the capacity of the machine) and threads should stay in their processor.

Disable the communication lines to have a better visualization of the workload. To disable the communications lines you should disable the toggle button COMM at the bottom of the displaying window and change its name to **Global view** (to change it, you have to change the Name text box and apply the changes). Without the communications lines we can see in which processors the applications have been executed.

Make zooms to see in more detail the trace file and works the communication lines (process migrations) work.

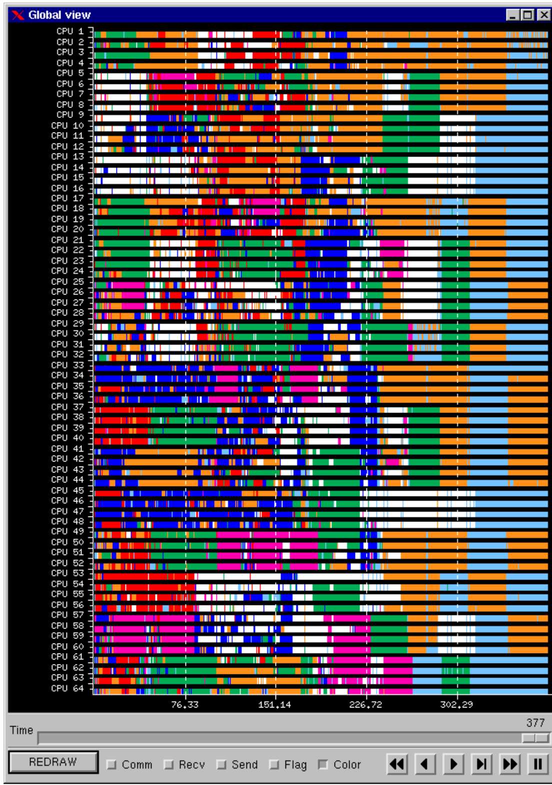


Figure 5.3: Processor allocation trace file (no migrations). Multiprogrammed executions

In next sections we are going to show how to collect statistics. We are going to collect statistics like : the parallelism obtained by each application versus his requested number of processors, how many time he has been executing in the same processor, how many process migrations has made an application, ...

As example, we use the processor allocation made by the operating system but any user can implement his/her own scheduler and study it using Paraver.

5.3 Study of a single application

One of the main improvements in Paraver is that it lets the user to select what he/she want to study to collect statistics. Using the different modules a lot of information could be extracted in different ways. For example, we could focus our study to all the workload or to one application.

In this section we are going to explain how Paraver can be used to focus our study to a single application and collect some statistics for an application.

When an application is launched, it requests a number of processors that want to use, but in a multiprogrammed environment there are other applications and not all the resources can be assigned to it, the operating system must distribute those resources between the different applications as well as possible.

Our workload request 96 processors distributed in different applications, but the machine only has 64, we requested more processors than physical so processor allocation policy could affect the workload execution time and the individual execution time for each application. A bad policy where there is a lot of process migration or a bad application grouping could affect the performance of the application.

For example, when using parallel applications usually there are many synchronization points where threads have to wait the threads that haven't finished their work yet. The scheduler must

share the resources because if it gives more time to a thread that was waiting the rest than other that has to finish their work, the application doesn't go on.

Focussing our analysis to one application will give a visualization of where it has been executed, how many migrations it has done, ... The next subsections will show a small analysis that can be done when working with one application.

5.3.1 Selecting one application.

To obtain a visualization where only one application is displayed, take as the current window the called window (*Global view*), go to the **Semantic Module** window and select at THREAD level the function called **Given State** (step **1** in figure 5.4). Select only the *Swim application* application by raising the DEFINED STATES window (to raise it, press the question mark button in the window to select the given states, step **2**) and select the Swim application state by clicking its toggle button (step **3**). Also, to make our study, we are only interested in the process migrations made by this application and we want to filter the rest. The process migrations has been coded as communications from processor X to processor Y and the communication tag gives the number of application which makes the migration.

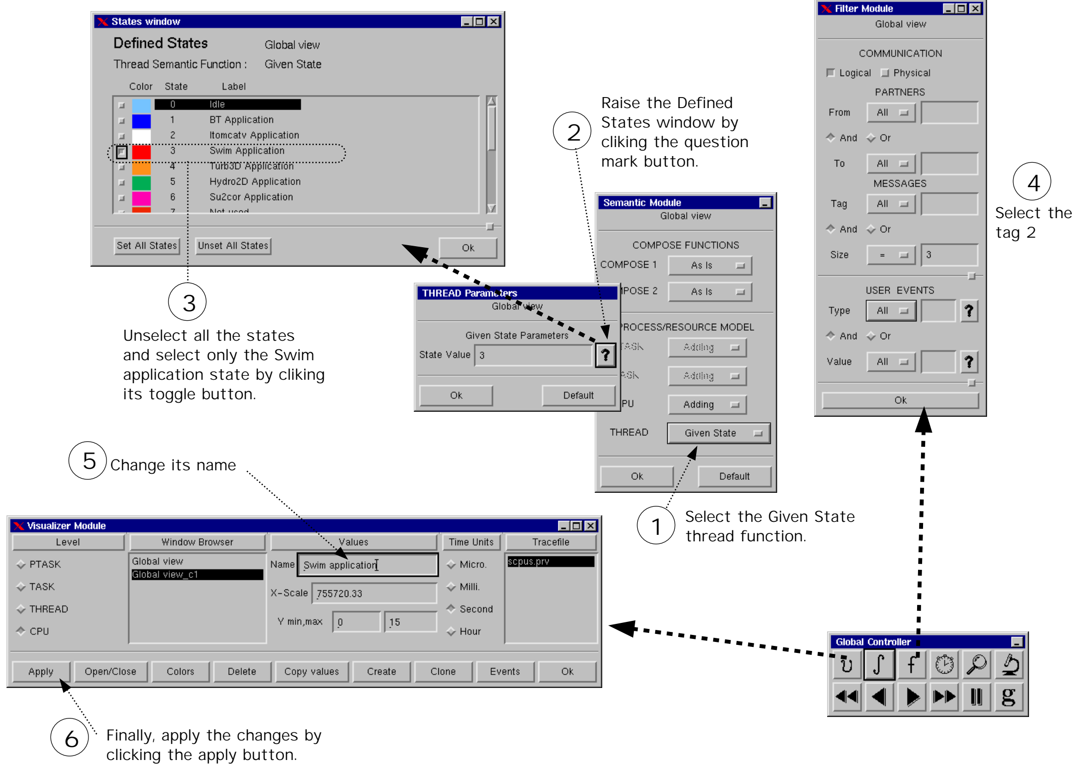


Figure 5.4: Selecting one application. Multiprogrammed executions

To filter it, go to the **Filter Module** and in the TAG line, select the symbol "=" and write a 3 in the text box next to the size pop up sub menu. This will filter all the process migrations (or communications) except those made by Swim application (state value 3).

Finally, before redrawing the window, enable the communication lines to draw it in the displaying window. As a result, we obtain a window like figure 5.5 where only one application is painted. You can see where it has been executed and the process migrations that have been done during the execution. Now, change its name to **Swim application**.

Figure 5.5 shows how the Swim application has been executed in all the processors in the machine, the visualization shows all the migrations done by the application threads.

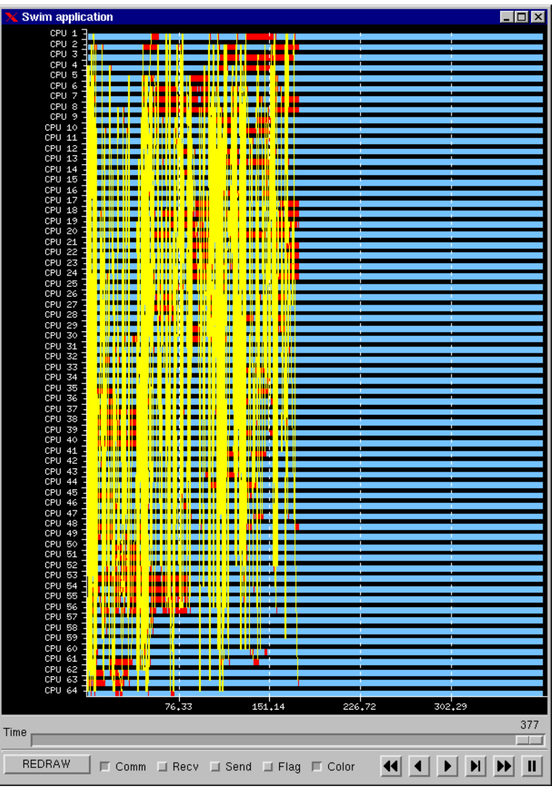


Figure 5.5: Swim application processor allocation. Multiprogrammed executions

5.3.2 Computing the number of process migrations and average execution time in each processor.

A first simple analysis can be done to compute the number of process migrations that the application has been done and the average execution time in the different processors. To obtain the processor migrations of the application we can use the functions which compute the number of sends (**# Sends**). The **Average Burst** function computes the average execution time of the application in each processor; the **Stdev Burst** function computes its standard deviation and the **# Bursts** function computes the number of times that it has been executed in that cpu.

To compute it make an **All trace** analysis for this window selecting the specified functions. As a result we obtain the information for each processor plus some lines presenting the sum, average, maximum and minimum values displayed.

The value for **# Sends** row shows the number of process migrations that this application has done during the workload execution (1023 process migrations).

5.3.3 Parallelism profile for each application during the execution.

Another study that could be done is the analysis of the average number of processors on the application has been executing during the workload versus the requested amount of processors. This analysis can be computed for any application within the workload and we obtain if the processor allocation used gives the number of processors requested by the application or on the other hand, the application has to run with less processors than requested.

To make our analysis, select the window which shows the swim execution (*Swim application*) and clone it to obtain an identical window. This window will be our working window to analyze the profile. To obtain a visualization of the parallel profile of the application, first, we have to change the **Given state** function to the **In state** function. The **In state** function returns the

selected states as a working state (state value 3 is returned as 1). Therefore, select the **In state** function at THREAD level, fill a 3 in its parameter window and redraw the window.

The new visualization draws a working state when the application is executed in a processor and an idle state otherwise (remember, that previous visualization draws the state as is). To obtain the profile, change the object representation level to PTASK level by selecting it in the Visualizer window. Before, applying the changes disable the communication lines from the displaying window and change the **Y max** scale to 16 (the application requested 16 processors so the maximum value for a parallelism profile will be 16). The figure 5.6 shows the application profile[3] obtained during the workload execution for Swim application after changing the scale and disabling the communication lines. Also, two zooms have been done at the beginning and at the end of the execution to select the initial and final points for the analysis.

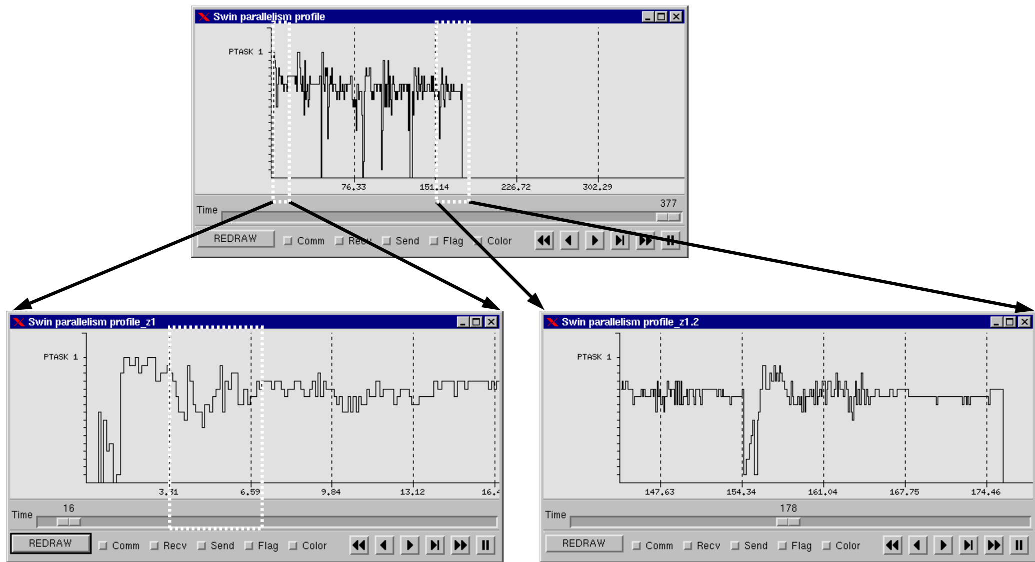


Figure 5.6: Profile zooms to make the analysis. Multiprogrammed executions

The profile visualization shows how sometimes the application runs with the requested processors but usually runs with less processors than requested. Also, the execution of the application finishes in the middle of the workload so to compute things like average number of processors where the application has been executing the ALL TRACE analysis doesn't work well because application finishes in the middle of the trace file, we should select the initial and final points of the application.

Make the analysis selecting the first point at the beginning of the trace file (window called **Swim Execution profile_z1**) and the second point at the ending of the trace file **Swim Execution profile_z2**) to fill all the Swim execution in the analysis limits.The **Avg Semantic Val** function gives the average number of processors during the execution. Note, that the result for Swim application is an average of 11.13 processors; application has requested 16.

Another results could be seen in this analysis, for example the **Average Burst** gives the average time that the application has been running with the same number of processors (in our example 212 milliseconds), the **# Burst** tells the number of times that the application has changed the number of processors so probably some threads hasn't been running (it has change 808 the number of processors).

To compute the analysis for the rest of applications you should take care because only until the swim finalization (is the first application which finishes their execution) there are all the

[3] Also, to capture the windows we have changed the background and foreground (using the option OPTIONS/SYSTEM COLORS). If you work normally with Paraver without changing any color, foreground will be the white color and background will be the black color.

application running with 96 requested processors, when some applications begins to finish, the rest that remains in execution have more free resources, so its requested processor number could be satisfied by the operating system (see figure ??) and its parallelism increases.

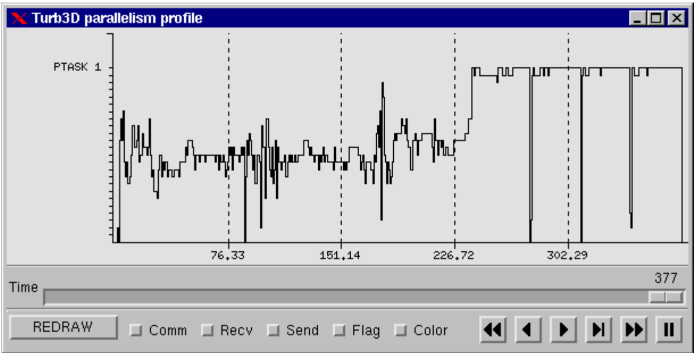


Figure 5.7: Turb3D profile. Multiprogrammed executions

Figure 5.7 shows the Turb3D parallelism profile, it has requested 24 threads but only an average of 12 has been executed when all the workload applications are running (so requested load is about 96 processors). At the end, when some applications have finished (thus, requested number is lower), its parallelism value is increased to 24.

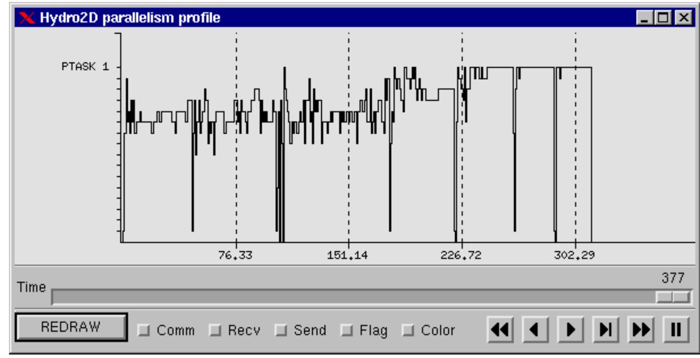


Figure 5.8: Hydro2D profile. Multiprogrammed executions

The Hydro2D (figure 5.8) has requested 16 threads but in its profile we can see that only an average of 10 threads are running at same time when all applications are active. As Turd3D application, when the requested lod goes down, its number of threads executing in parallel is increased.