Paraver

Version 3.0

Parallel Program Visualization and Analysis tool

TRACEFILE DESCRIPTION

June 2001

Contents

List of Figures

Chapter 1

Introduction

Paraver is a flexible parallel program visualization and analysis tool based on an easy-to-use Motif GUI. Paraver was developed responding to the basic need of having 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.

Some Paraver features are the support for :

- Detailed quantitative analysis of program performance
- Concurrent comparative analysis of multiple traces
- Fast analysis of very large traces
- Mixed support for message passing and shared memory (networks of SMPs)
- Easy personalization of the semantics of the visualized information

One of the main features of Paraver is the flexibility to represent traces coming from different environments. Traces are composed of state transitions, events and communications with an associated timestamp. These three elements can be used to build traces that capture the behaviour along time of very different kinds of systems. The Paraver distribution includes, either in its own distribution or as an additional package, the following trace generators:

1. Sequential application tracing: it is included in the Paraver distribution. It can be used to trace the value of certain variables, procedure invocations, ... in a sequential program.
2. Parallel application tracing: a set of modules are optionally available to capture the activity of parallel applications using shared–memory (OpenMP directives), message–passing (MPI library) paradigms, or a combination of them.
3. System activity tracing in a multiprogrammed environment: an application to trace processor allocations and migrations is optionally available in the Paraver distribution.

This document includes a detailed description of the Paraver programming model and trace format. This allows Paraver users to develop their own tracing facilities according to their own interests and requirements. The possibilities offered by the visualization, semantic and quantitative analyzer modules are powerful enough allowing users to analyze and understand the behaviour of the traced system. Paraver also allows the customization of some of its parts as well as the plug–in of new functionalities.

So expressive power, flexibility and the capability of efficiently handling large traces are key features addressed in the design of Paraver. The clear and modular structure of Paraver plays a significant role towards achieving these targets. Let us briefly describe the key design philosophy behind these points.

Views

Paraver offers a minimal set of views on a trace. The philosophy behind the design was that different types of views should be supported if they provide qualitatively different analysis types of information. Frequently, visualization tools tend to offer many different views of the parallel program behaviour. Nevertheless, it is often the case that only a few of them are actually used by users. The other views are too complex, too specific or not adapted to the user needs.

Paraver differentiates three types of views :

- Graphic view : to represent the behaviour of the application along time in a way that easily conveys to the user a general understanding of the application behaviour. It should also support detailed analysis by the user such as pattern identification, causality relationships, ...
- Textual view : to provide the utmost detail of the information displayed.
- Analysis view: to provide quantitative data.

The **Graphic View** is flexible enough to visually represent a large amount of information and to be the reference for the quantitative analysis. The Paraver view consists of a time diagram with one line for each represented object. The types of objects displayed by Paraver are closely related to the parallel programming model concepts and to the execution resources (processors). In the first group, the objects considered are : *application* (Ptask in Paraver terminology), *task* and *thread.* Although Paraver is normally used to visualize a single application, it can display the concurrent execution of several applications, each of them consisting of several tasks with multiple threads.

The information in the graphics view consists of three elements : a time dependent value for each represented object, flags that correspond to puntual events within a represented object, and communication lines that relate the displayed objects. The visualization control module determines how each of these elements are displayed. The essential choices are :

- Time dependent value : displayed as a function of time or encoded in color. Furthermore, time and magnitude scale can be changed to zoom the visualization.
- Flags : displayed or not and color.
- Communication : displayed or not.

The visualization module blindly represents the values and events passed to it, without assigning to them any pre-conceived semantics. This plays a key role in the flexibility of the tool. The semantics of the displayed information (activity of a thread, cache misses, sequence of functions called,...) lies in the mind of the user. Paraver specifies a trace format but no actual semantics for the encoded values. What it offers is a set of building blocks (semantic module) to process the trace before the visualization process. Depending on how you generate the trace and combine the building blocks, you can get a huge number of different semantic magnitudes.

Expressive power

The separation between the visualization module which controls how to display data and the semantic module which determines the value visualized offers a flexibility and expressive above than frequently encountered in other tools.

Paraver semantic module is structured as a hierarchy of functions which are composed to compute the value passed to the visualization module. Each level of function corresponds to the hierarchical structure of the process model on which Paraver relies. For example: when displaying threads, a thread function computes from the records that describe the thread activity, the value to be passed for visualization; when displaying tasks, the thread function is applied to all the threads of the task and a task function is used to reduce those values to the one which represents the task; when displaying processors, a processor function is applied to all the threads that correspond to tasks allocated to that processor.

Many visualization tools include a filtering module to reduce the amount of displayed information. In Paraver, the filtering module is in front of the semantic one. The result is added flexibility in the generation of the value returned by the semantic module.

Combining very simple semantic functions (sum, sign, trace_value_as_is,....), at each level, a tremendous expressive power results. Besides the typical processor time diagram, it is for example possible to display:

- The global parallelism profile of the application.
- The total per CPU consumption when several tasks share a node.
- Average ready queue length of ready tasks when several tasks share a node.
- The instantaneous communication load geometrically averaged over a given time.
- The evolution of the value of a selected variable, ...

The default filtering and semantic function setting of the tool results in the same type of functionality of many other visualization tools. A much higher semantic potential can be obtained with limited training.

Direct and Derived metrics

Paraver offers a very flexible mechanism to compute and display a large number of performance indices and derived metrics from the trace.

There is a first level of semantic functions which obtain the values in function of time directly from the records in the trace. A second level of semantic functions can be obtained by combining (add, divide, ...) the functions of time computed by the first level.

For example, starting with a window that looks at the cycles counter and another window that looks at the instructions counter, it is possible to apply the second level of semantic functions in order to obtain a derived metric like Instructions per Cycle by dividing those two windows. Although the trace doesn't contain any counter about IPC, it could be computed from the cycles counter and instructions counter.

Quantitative analysis

Global qualitative display of application behaviour is not sufficient to draw conclusions on where the problems are or how to improve the code. Detailed numbers are needed to sustain what otherwise are subjective impressions.

The quantitative analysis module of Paraver offers the possibility to obtain information on the user selected section of the visualized application and includes issues such as being able to measure times, count events, compute the average value of the displayed magnitude,...

The quantitative analysis functions are also applied after the semantic module in the same way as the visualization module. Again here, very simple functions (average, count, ...) at this level combined with the power of the semantic module result in a large variety of supported measures. Some examples are:

- Precise time between two events (even if they are very distant)
- Number of messages sent between time X and Y

- Number of messages of tag T exchanged between processor P1 and P2 between time X and Y
- Average CPU utilization between time X and Y
- Number of events of type V on processor W between time X and Y
- Average CPU time between two communications

Multiple traces

In order to support comparative analyses, simultaneous visualization of several traces is needed. Paraver can concurrently display multiple traces, making possible to use the same scales and synchronized animation in several windows.

This multi-trace feature supports detailed comparisons between:

- Two versions of a code
- Behavior on two machines
- Difference between two runs
- Influence of problem size

Comparisons which otherwise are very subjective or cumbersome.

Customization

Developing a tool which fulfills the needs of every user is rather impossible. Initially Paraver aimed at supporting the projects carried out at **CEPBA** as part of our research and service activities. One objective of the design was to provide some support for expert users having new needs and willing to extend the functionalities of Paraver. For this purpose, Paraver is distributed with the possibility of personalizing the two modules that provide the expressive and analysis power.

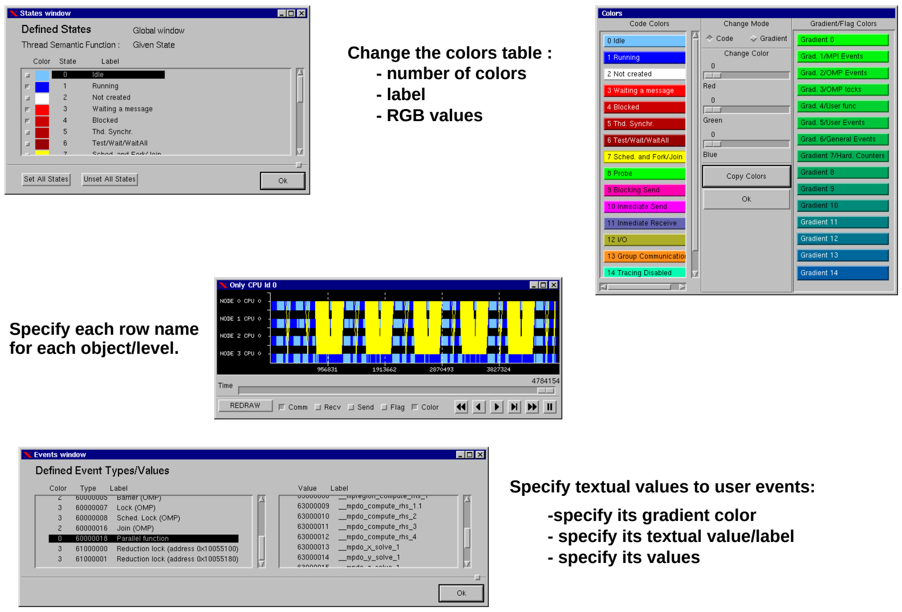

Change the colors table :
- number of colors
- label
- RGB values
Specify each row name for each object/level.
Specify textual values to user events:
-specify its gradient color
- specify its textual value/label
- specify its values

Figure 1.1: Paraver Configuration File

A procedural interface is provided in such a way that a user can, with a limited effort, link into its Paraver version the actual functionality needed. Taking into account the built in semantic functions and analysis functions and their relation to the hierarchical process model and the possibility of combining them in a totally orthogonal way at each level, a user can obtain a large number of new semantics by developing very simple modules.

Another aspect where the users may have personal preferences is in setting color tables. More important is the possibility to specify the textual values associated with the values encoded in the trace. All this is achieved through a simple but powerful **configuration file** (see Appendix 4).

Configuration files are simple, but very useful files which lets to the user customize his/her environment, save/restore a session, ... Also, we can change some default options and redefine the environment.

It is important to know and use these files because they make easier the day work with Paraver.

Large traces

A requirement for Paraver was that the whole operation of the tool has to be very fast in order to make it usable and capable to maintain the developer interest. Handling traces in the range of tenths to hundreds of MB is an important objective of Paraver (see chapter 3 **Paraver Trace file**).

Easy window dimensioning, forward and backward animation, zooming are supported. Several windows with different scales can be displayed simultaneously supporting global qualitative references and detailed analysis. Even on very large traces, the quantitative analysis can be carried out with great precision because the starting and end points of the analysis can be selected on different windows.

Chapter 2

Paraver Object Model

As described in the introduction, Paraver functionality is tightly coupled with the hierarchical object model targeted by the DiP environment. Paraver works with two ortogonal object models:

- The **process model** is composed by the objects that correspond to the three levels of the most frequently programming models: application, process and thread objects.
- The **resource model** represents the physical resources where the different threads are finally executed. The resource objects are tightly connected with the cluster of SMPs where applications has been executed: processor and node.

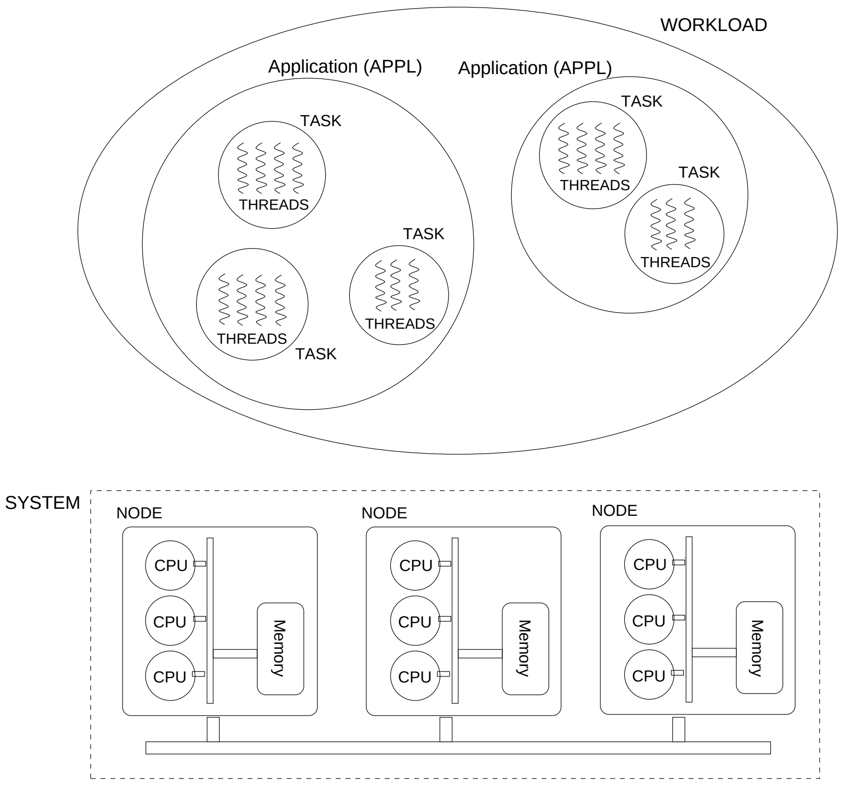


Figure 2.1: Paraver object model

To exploit all the object model levels, the tools that generates trace files for Paraver must be capable to get the process and resource information. Both shared and distributed memory applications could be mapped on the Paraver object model.

Object Model flexibility

The Paraver **Process and Resource models** define the structure of two ortogonal type of objects for which performance indices can be computed and displayed. The actual names used for those objects derive from the standard parallel programing terminology. Nevertheless. Paraver does not assume any semantics beyond the hierarchical structure of the objects. It is possible for any user to implement instrumentation or simulation tools that map other concepts onto the thread, cpu, task, ... names. For example, the resources model could be used to represent functional units and threads could represent instruction flow. Features such as the behaviour of cluster arquithectures could be easely displayed.

2.1 Paraver Process Model

This process model is a superset of the most frequently used programming models. On a Paraver window, the type of process model object to be represented can be selected among:

- Set of applications (WORKLOAD)
- Application (APPL)
- TASK
- THREAD

A **parallel application** (APPL level) is composed by a set of sequential or parallel **processes** (named as TASK in paraver process model hierarchy). The parallel processes are composed by more than one thread while the sequential processes are mapped into one thread. The top of the process model hierarchy is the WORKLOAD level representing a set of different applications running on the same resources.

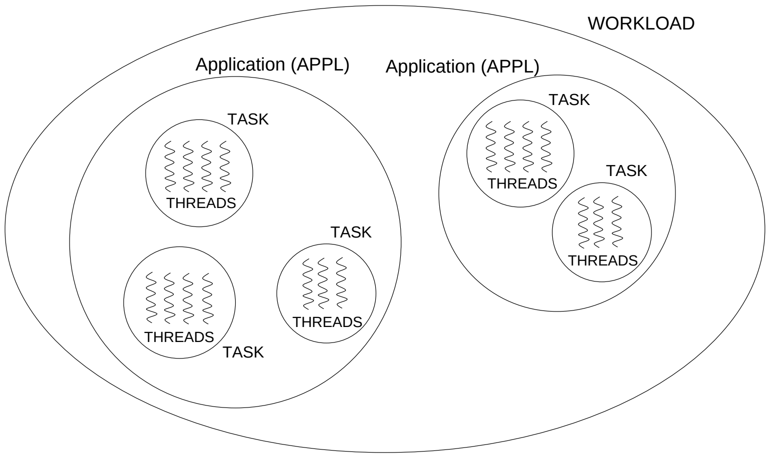


Figure 2.2: Paraver process model

This model is very flexible, and can be easily mapped to the models supported by actual communication or multi-threaded libraries.

A thread type window will display one line for each selected thread. Paraver supports several applications concurrently running on the same system thus, on an application type window, one line will be displayed for each application.

In the three-level process model an application can have one or several tasks, and each task can have one or several threads. Tasks do not share address space, thus communication between them is only done through message passing. The different threads within a task are executed within the address space defined by the task. Threads within a task can thus communicate and synchronize through shared memory.

The value represented on a line of a given type is computed from the records in the trace by the hierarchical composition of functions corresponding to each one of the levels in the model (see figure 2.3). Top levels semantic functions return a value based on the semantic values returned by the bottom levels. For example, the semantic value returned for TASK level is computed based on the semantic values of the threads of the task.

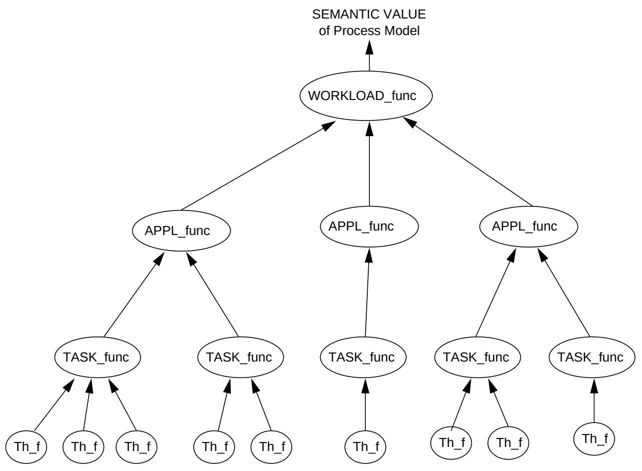


Figure 2.3: Hierarchical composition of levels

Shared and distributed memory programming models can be mapped onto the *Paraver process model.* For example, we can map the MPI programming model, the PVM programming model, the OpenMP programming model, combined MPI and OpenMP programming models, ...

Example 1: Mapping the MPI programming model.

The mapping of the **MPI programming model** onto the *Paraver process model* can be done as follows :

- each MPI process is a Paraver TASK with one Paraver THREAD
- the whole MPI application is the Paraver APPL (application object), grouping all MPI processes.

This mapping lets to have in a tracefile the execution of different MPI applications at the same time.

The **PVM programming model** can be mapped like the **MPI programming model**. Each PVM process will be a Paraver TASK.

Example 2: Mapping the combined OpenMP+MPI programming model.

The combined OpenMP+MPI programming model could be seen as a MPI application where each MPI process is composed by a set of threads which work in parallel. The mapping that could be done is a combined mapping between the two programming models :

- each OpenMP thread is mapped on one Paraver THREAD
- each MPI process composed of multiple OpenMP threads is a Paraver TASK

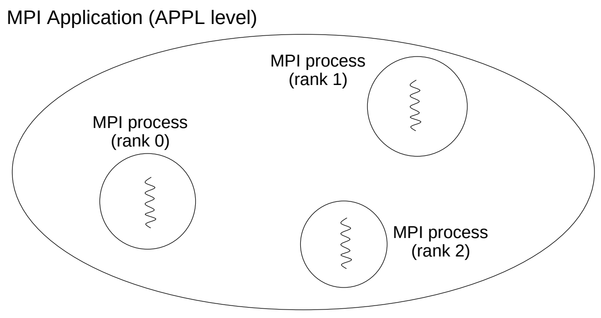


Figure 2.4: Mapping of the MPI programming model

- the whole OpenMP+MPI application is the Paraver APPL (application object).

As in the previous example, we can have a trace with different applications.

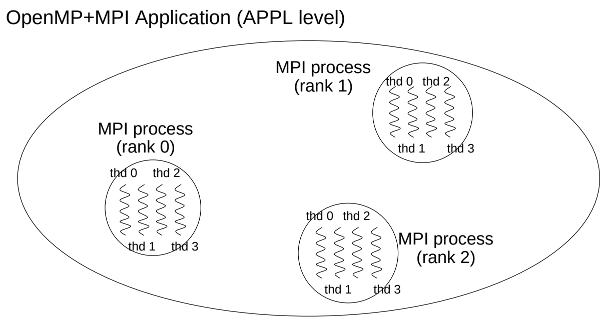


Figure 2.5: Mapping of the OpenMP+MPI programming model

2.2 Paraver Resouce Model

The resource model represents the resources where the applications are executed. On a Paraver window, the type of resource object to be represented can be selected among:

- Cluster of nodes (SYSTEM)
- NODE (set of processors)
- Processor (CPU)

The processors are the resources where the threads are executed. Processors are grouped in nodes. A task is mapped into a node and thus all its threads share their processors in that node.

The mapping is not necessarily one to one, so it is possible to have several tasks from a single application on a given node. Tasks from different applications can also be mapped into the same node.

Different resource models can be mapped onto the *Paraver resource model*. For example, a uniprocessor machine could be represented by a single node composed by one processor. A single shared memory multiprocessor with four processors could be represented as one node composed by four processors. A distributed shared memory could be represented by different nodes could be mapped by more than mode composed by different processors. At the top, the **system** level has been added in order to represent a set of SMP clusters.

For a processor type window, the valued displayed is the value returned by the thread funtion that is executing in that moment.

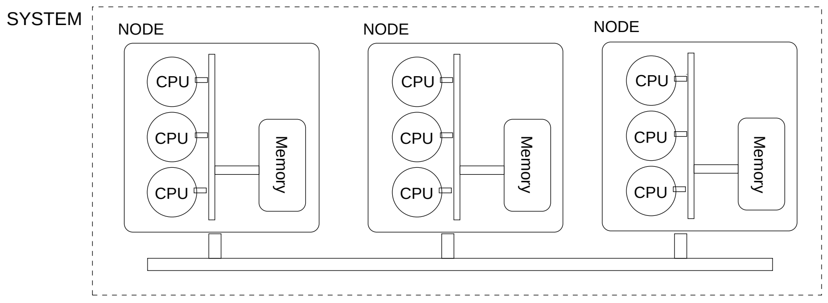


Figure 2.6: Paraver resource model

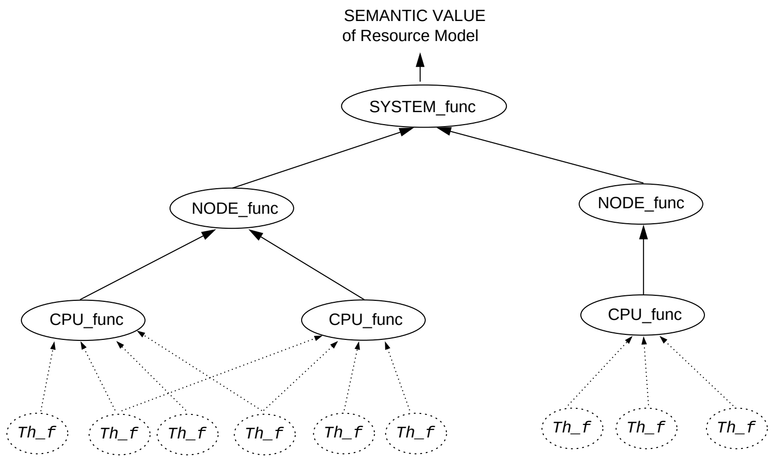


Figure 2.7: Hierarchical composition of levels

The semantic function for one node is computed based on the semantic values of the processors of that node. Typical combinations are addition, average, maximum, ... New processor semantic functions such as Active Thread or Active Thread Value have been developed that support new types of analyses. These functions return the **identifier** or **value** respectively of the thread running on the processor at the moment.

Chapter 3

Paraver Trace file

The trace file contains records which describe absolute times when events or activities take place during the execution of the parallel code. Each record represents an event or activity associated to one thread in the system. Furthermore, trace files have associated some other files to configure some aspects of the environment: number of states, color and state labels, user event labels, row labels ...

The three basic types of records defined in Paraver are :

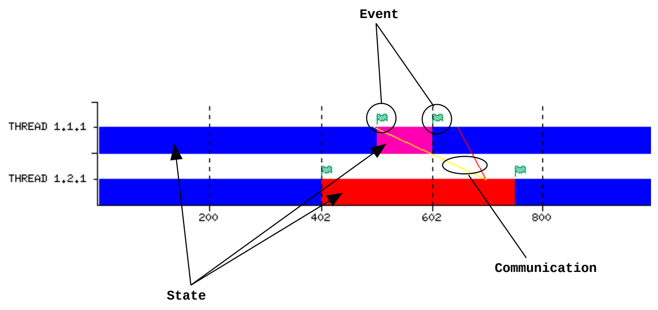


Figure 3.1: Paraver record types

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

Event records represent punctual events in the code. These 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.

An important issue in the tracing implementation must be precision and accuracy in the measurement. By default, the trace file works with microsecond precision and to take benefit of it, the tracing tools should be capable to get the events and activities at microsecond presicion. However, if tracing tool works with lower presicion (like *nanoseconds* or *cycles*), they also could be assigned taking in mind that lowest units when working inside Paraver will be that ones (even though unit labels could be changed).

A Paraver trace could be composed by three files :

- the **ASCII trace file** which defines the objects structure and contains the list of all the trace records (states, events and communications). The ASCII trace file name usually ends using the extension **.prv**.
- the **Paraver Configuration File** which defines the labels and colors associated to states and events. When a trace is loaded, if there is a file named like ASCII trace name but with the **.pcf** extension instead of **.prv**, Paraver loads it automatically.
- the **Names Configuration File** which defines the row labels that will be used instead default ones. By default, each row label is composed by the level and its object identifiers (for example, the row labels when working at thread level are composed by the application identifier, the task identifier within the application and the thread identifier within the task i.e. THREAD 1.2.3). This file lets to define user row labels. The user could define the label NAS LU MASTER instead of label THREAD 1.1.1. When a trace is loaded, if there is a file named like the trace file but with the **.row** extension instead of **.prv**, Paraver loads it automatically.

Although, Paraver and Names Configuration files are optional, we mainly recomend to use it, specially the Paraver Configuration File which defines labels and colors to states an events. Tracing tools should generate them together with the trace file.

The rest of the chapter will describe the ASCII trace format and some important features that tools must take in mind when generating trace files. Later, the chapter 4 on page 23 will describe the two configuration files that should be generated together with the trace file.

3.1 ASCII Trace Format

The ASCII trace format is composed by a header and a body (see figure 3.2). The header describes the process and resource model objects and the body contains the ordered list of records.

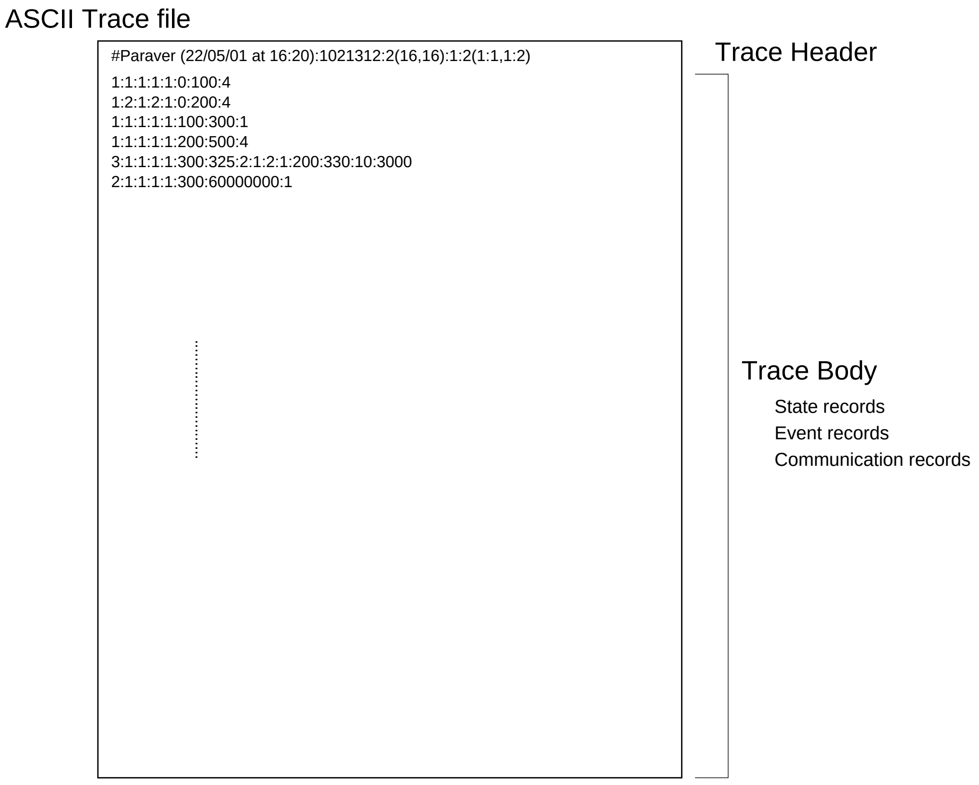
3.1.1 Paraver trace header

The trace header defines the process and the resource model of the tracefile. It contains the information about the number of applications of the tracefile, the number of tasks for each application and the number of threads in each task. Furthemore, the header defines the number of nodes and its number of processors.

The trace header is a line where the different fields define the object structure (fields are separated by colons). The trace header format is :

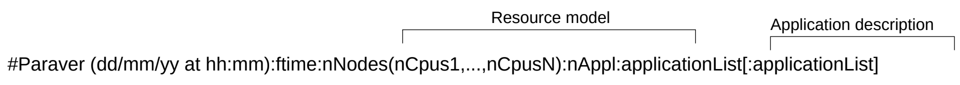
- **#Paraver (dd/mm/yy at hh:mm):** defines the date and hour where trace has been generated. Is is important to use the symbol # at the beginning of the header line because it inidicates that it is in ASCII trace format.
- **ftime:** total trace time in microseconds
- **nNodes(nCpus1[,nCpus2,...,nCpusN]):** defines the number of nodes and number processors per node. After the number of nodes (**nNodes**), the list of the number of processors must be specified (**nCpus1** is the number of processors on node 1, **nCpus2** is the number of processors on node 2,...).
- **nAppl:** number of applications in the trace file.
- **applicationList :** The application list defines the application object structure. Each application has its application list (applicationList) separared by a colon. The application list format is:

```
nTasks(nThreads1:node,...,nThreadsN:node)
```



```
#Paraver (22/05/01 at 16:20):1021312:2(16,16):1:2(1:1,1:2)
1:1:1:1:1:0:100:4
1:2:1:2:1:0:200:4
1:1:1:1:1:100:300:1
1:1:1:1:1:200:500:4
3:1:1:1:1:300:325:2:1:2:1:200:330:10:3000
2:1:1:1:1:300:60000000:1
```

Figure 3.2: ASCII trace structure



```
#Paraver (dd/mm/yy at hh:mm):ftime:nNodes(nCpus1,...,nCpusN):nAppl:applicationList[:applicationList]
```

Figure 3.3: Header format

where **nTasks** is the number of tasks of the application.The list defines for each task, the number of threads (**nThreads1**) and the node where it has been executing (**node**).

Traces without resource information

Not all tracing tools may be capable to get the resource information during the execution because system doesn't give this type of information or the tool doesn't know how to obtain it.

If the resource object levels are not use, it does not make sense to define it in the trace file header. To disable the resource levels, a zero number must be used when defining the resource information. When Paraver loads a trace without resource levels, they are disabled.

```
#Paraver (dd/mm/yy at hh:mm):ftime:0:nAppl:applicationList[:applicationList]
```

Example of a trace file header

We are going to construct the trace header corresponding to the object structure shown in figure 3.4. The tracefile will have two applications that have been executed in a cluster of two SMPs.

- **Process model objects:** The first application is based on a mixed OpenMP+MPI programming model, it is composed by two MPI processes with four OpenMP threads. The second one is an OpenMP application running with four threads.

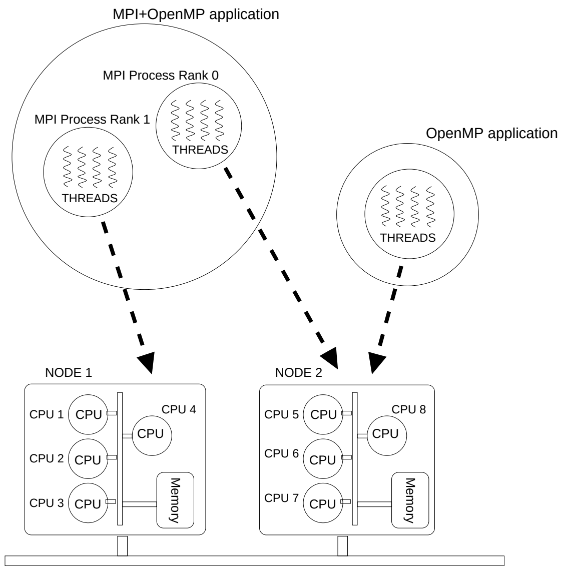


Figure 3.4: Object structure

- **Resource model objects:** The two applications has been executed in a cluster of two SMPs with four processors.

As we showed in chapter 2, the mapping of the OpenMP+MPI programming model could be done as follows:

- each OpenMP thread is mapped on one Paraver THREAD
- each MPI process composed of multiple OpenMP threads is a Paraver TASK
- the whole OpenMP+MPI application is the Paraver APPL (application object).

thus, on the process object model, we have one APPL with two TASKs and each of them composed by four THREADs. The OpenMP programming model could me mapped in one APPL composed by a single TASK with four THREADs. On the resource object model, we could map the two clusters on 2 NODEs composed by four CPUs.

The header must describe this objects structure, the process and resource model and should look like:

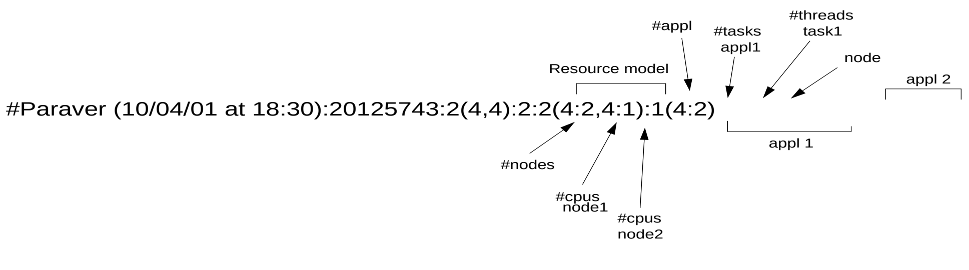


Figure 3.5: Header Example

The resource model is defined by **:2(4,4):**, where the number of nodes and the number of processors per node is specified.

The object model is defined by **:2:2(4:2,4:1):1(4:2)**. First, it states the number of applications (**:2:**), and for each application there is a list that describes its structure (**:2(4:2,4:1):1(4:2)**).

3.1.2 Paraver trace body

The trace body contains an ordered list of records. Paraver trace has three types of records: states, user events and communications.

The trace records represent state of a thread, an event on a punctual time or communication (relationship) between a couple of threads. Each of them is identified by a different type. State records represent an activity associated to one thread in the system that happens in a specific resource while event records represent an event associated to one thread at a certain time. Communication rrecords represent relationships between two threads at a certain time.

Since a single trace file may contain several applications, records are tagged with the application number, the task number within the application where thread belongs and the thread identifier within the task and the resource identifier (processor number) where it is associated.

All processors from all nodes are numbered by a single global identifier which will identify the processors from different nodes (figure 3.4 on page 16 shows how processors are numbered). It is important to node that processors numbering goes from first node to last one. Using this numbering, we can easely identify the processor identified as 7 in figure 3.4 as the fourth processor of second node.

State record

State records represent intervals of actual thread status. The first field is the record type identifier (for state records, type is 1). The next fields identify the resource (cpu_id field) and the object to which the record belongs (appl_id, task_id and thread_id fields). Remember that cpu_id is the global processor identifier (if no resource levels have been defined the processor identifier must ever be a zero). Beginning time and ending time of the state record also have to be specified, and finally, the state field is an integer that encodes the activity carried out by the thread.

```
1:cpu_id:appl_id:task_id:thread_id:begin_time:end_time:state
```

If state is not assigned to any processor, its cpu_id should be set to zero. For example, in next state records :

```
#Paraver(23/02/01 at 18:57):500:1(2):1:1(1:1)
1:2:1:1:1:0:200:1
1:0:1:1:1:200:300:1
1:1:1:1:1:300:500:1
```

the state value 1 of thread 1.1.1 in first record has been using processor 2 from time 0 to 200, the second one has not been using any processor from time 200 to 300, and third one has been using the processor 1 from time 300 to 500.

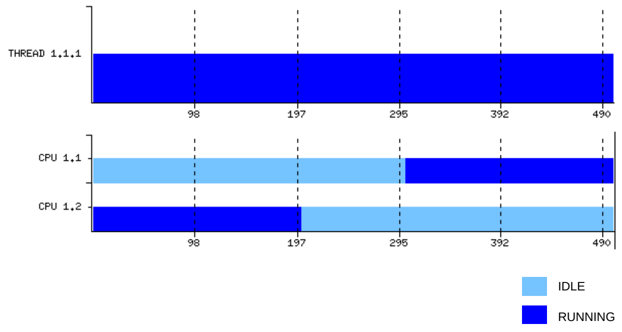


Figure 3.6: Thread view vs. processor view. State record

Figure 3.6 shows the State As Is visualization of the three state records at thread and cpu level. Dark blue color corresponds to the state value 1 (by CEPBA-UPC tracing tools it is considered as running state) while light blue color corresponds to the state value 0 (considered as idle state). Note that at thread view a long running burst is shown (state value 1) while at the processor view there is a first burst on processor 2, no bursts at any processor and finally, a burst in processor 1. Although at thread view all time is considered as running state, from time 200 to 300, thread in figure 3.6 has not been running at any processor (it could be considered that it has been descheduled)

- **Important: Not overlap threads on the same processor**

  Each record represents a thread and the resource where it was performed. The tracing tool must garantiee that two records will not use the same resource at the same time. It is needed to obtain a correct trace visualization at resource levels.

  To show it, we are going to see an erroneus trace example that overlaps thread states in the same processor and after, we will show an example of the correct trace that should be generated.

  Our first trace example is composed by two threads sharing the same processor. The **Running** state (state value 1) cosumes processor time (its cpu_id is the processor number) and the **Waiting for CPU** state (state value 5) doesn't cosume processor time (its cpu_id is the zero value).

  Trace is:

```
#Paraver(23/02/01 at 18:57):200:1:1:1(2:1)
1:1:1:1:1:0:75:1
1:0:1:1:2:0:60:5
1:1:1:1:2:60:170:1
1:0:1:1:1:75:150:5
1:1:1:1:1:150:200:1
1:0:1:1:2:170:200:5
```

  Figure 3.7 shows its visualization of the previous trace example. The first window shows the thread states: dark blue is the Running state (state value 1) and red is the Waiting for CPU state (state value 5). The second window shows the processor view, the colors displayed represent the thread identifier that is executing. Here, the dark blue color is the first thread (value 1) and the white color (value 2) is the second one.

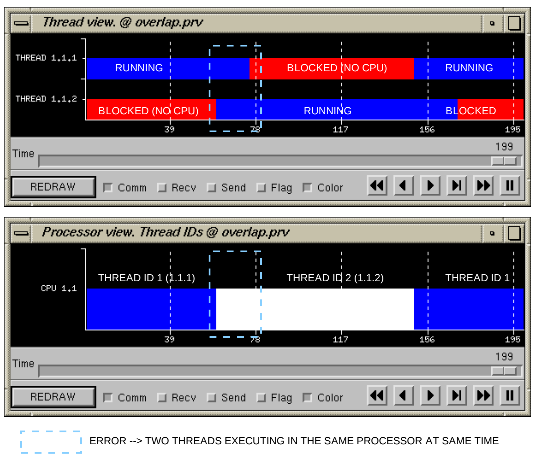


Figure 3.7: Erroneus overlaped trace.

  Note that Running states are overlapped. Thread 1.1.2 begins to run while thread 1.1.1 is still executing and later, while thread 1.1.2 is executing, the thread 1.1.1 begins its execution on the processor another time. *It will mean that the two threads have been executing at the same time on the same processor which is incorrect to the processor model defined by Paraver and sometimes displayed information could be erroneus.*

The correct trace should be:

```
#Paraver(23/02/01 at 18:57):200:1:1:1(2:1)
1:1:1:1:1:0:75:1
1:0:1:1:2:0:75:5
1:0:1:1:1:75:150:5
1:1:1:1:2:75:150:1
1:1:1:1:1:150:200:1
1:0:1:1:2:150:200:5
```

where no threads overlaps on at the same time the processor. When first thread is descheduled, the second begins its **Running** burst that consumes processor.

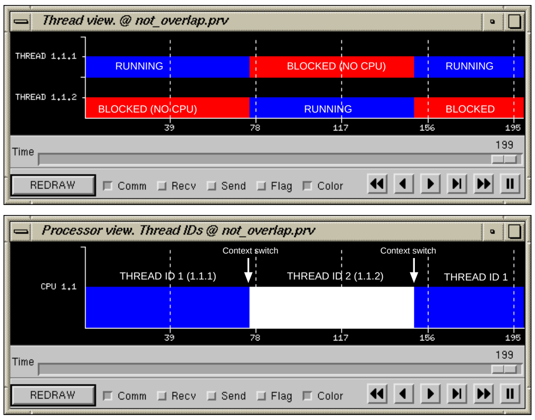


Figure 3.8: Correct trace.

Figure 3.8 shows the visualization of the previous trace. As with the previous trace, the first window shows the thread states: dark blue is the Running state (state value 1) and red is the Waiting for CPU state (state value 5). The second window shows the processor view, the colors displayed represent the identifier of the running thread. Here, the dark blue color is the first thread (value 1) and the white color (value 2) is the second one. Note that two thread Running states aren't overlapped.

Usually, the different defined states encode the different execution activities. Next table 3.1 shows the different activities considered by **OMPItrace tool** (CEPBA-UPC tracing tool used to trace OpenMP/MPI applications) where thread could be performing during an application execution:

The tracing tools could define their own states meaning, but it is recommended to adjust these new states to similar defined activities (see for example, the defined states in OMPItrace tool in table 3.1). For example, synchronizations are usually shown in red, the running state is shown as dark blue instead of idle state which is shown as light blue, overheads as yellow, ...

Event record

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

State id.	Activity description	Label
0	Thread is in an idle loop waiting for more work	Idle
1	Running application code	Running
2	The thread hasn't been created yet	Not created
3	Thread is waiting a message from another thread	Waiting a message
4	Thread is blocked	Blocked
5	Thead is is in a synchronization point (getting a lock, barrier, ...)	Thd. Synchr.
6	Thead is doing message passing operations like MPI_Wait,...	Wait/WaitAll
7	Thead is doing library code	Sched. and Fork/Join
8	Thead is doing message passing operations like MPI_Test, ...	Test/Probe
9	Thread is sending a message (blocking send)	Blocking Send
10	Thread is sending a message (non-blocking send)	Immediate Receive
11	Thread is waiting a message (non-blocking receive)	Immediate Receive
12	Thread is doing an I/O operation	I/O
13	Thread is doing an global MPI operation	Global OP
14	Tracing has been disabled	Tracing Disabled

Table 3.1: Example: Some thread activities during an execution

```
2:cpu_id:appl_id:task_id:thread_id:time:event_type:event_value
```

The first field is the record type identifier (for event records, type is 2). The next fields identify the resource (cpu_id field) and the object where record belongs (appl_id, task_id and thread_id fields). Then, the current absolute time of the event occurrence is specified. Event type and event value are also specified. The type identifies the event and the value is used to distinguish events of the same type.

If a zero value is assigned to the cpu_id, the event will not be displayed in resource model views (CPU,NODE and SYSTEM). For example, in next trace:

```
#Paraver(23/02/01 at 18:57):500:1(2):1:1(1:1)
1:2:1:1:1:0:100:1
2:2:1:1:1:100:5000:1
1:2:1:1:1:100:200:1
1:0:1:1:1:200:300:1
1:1:1:1:1:300:400:1
2:0:1:1:1:400:5000:0
1:1:1:1:1:400:500:1
```

the first event cpu_id (type=5000, value=1) has been assigned to number 2 but the second event cpu_id (type=5000, value=0) has not been assigned to any processor. Thus, as figure 3.9 shows, the second user event is not displayed at processor view but at thread view it is.

Communication record

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.

```
3:object_send:lsend:psend:object_recv:lrecv:precv:size:tag
```

- **object_send:** cpu_send_id:ptask_send_id:task_send_id:thread_send_id
- **lsend:** absolute time indicates when the user wants to send a message
- **psend:** absolute time indicates when the message is really sent

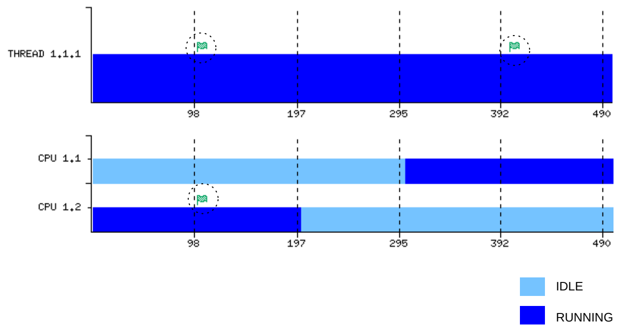


Figure 3.9: Thread view vs. processor view. Event record

- **object_recv:** cpu_recv_id:ptask_recv_id:task_recv_id:thread_recv_id
- **lrecv:** absolute time indicates when the user wants to receive a message
- **precv:** absolute time indicates when the message is really received
- **size:** integer greater than zero. It represents the size in bytes of the message
- **tag:** integer greater than zero. It is the message type identifier

Communications will not be displayed in resource model views (CPU,NODE and SYSTEM) if a zero value is assigned to the send and receive cpu_id.

3.1.3 Paraver trace order

```
rec_type:cpu_id:appl_id:task_id:thread_id:time:...
```

A Paraver trace file must have a certain order:

1. Ascending order of **time** (*logical send time* for communication)
2. Descending order of **record type** for records with same time. Thus records with same time must be ordered as: first communication traces (type=3), second event traces (type=2) and finally the sate traces (type=1).

3.1.4 Trace examples

Trace example (Resource model has not been defined)

Next example shows a bit of the beginning of a real Paraver trace file where the resource model has not been defined:

```
#Paraver (10/04/01 at 18:21):620244:0:1:1(4:0)
2:0:1:1:1:0:40000001:1
1:0:1:1:1:0:5124:1
1:0:1:1:2:0:14981:0
1:0:1:1:3:0:5568:0
1:0:1:1:4:0:6209:0
2:0:1:1:1:5124:60000019:2
1:0:1:1:1:5124:5139:1
2:0:1:1:1:5139:60000001:1
1:0:1:1:1:5139:5400:7
2:0:1:1:1:5400:60000018:1
```

```
    1:0:1:1:1:5400:162283:1
    2:0:1:1:3:5568:60000018:1
    1:0:1:1:3:5568:158294:1
    2:0:1:1:4:6209:60000018:1
    1:0:1:1:4:6209:146708:1
    ...
```

Note, that the resource model information in the trace header has been set to zero. In this trace file there isn't any information about where threads have been executing so the **processor identifier for all the trace records must be a zero value** because the tracing tool has not been able to get the resource allocation during the execution.

The header give us information about the date (10/04/01) and time (18:21) of the execution. The total application time is 620244 microseconds.

Trace example (Resource model has been defined)

Next example shows the beginning of a real Paraver trace file obtained from the execution of a SWIM application by on an IBM SP2. The tracing has been done using the **PE benchmarker tool** (tool from the *IBM Parallel Environment for AIX*).

The *PE Benchmarker* generates an UTE trace. The obtained UTE trace contains information about resource allocation during the execution and it has been translated to Paraver format using the UTE2Paraver tool (CEPBA-UPC tool to translate UTE trace format to Paraver trace format) to exploit all the resource information.

```
#Paraver(31/10/00 at 17:44):20156361:2(8,8):1:2(10:1,10:2)
1:0:1:1:1:0:5539:5
1:0:1:1:2:0:7039:5
1:0:1:1:3:0:161783:5
1:0:1:1:4:0:403127:5
1:0:1:1:5:0:403127:5
1:0:1:1:6:0:403127:5
1:0:1:1:7:0:403127:5
...
1:3:1:1:1:5539:10097:1
1:1:1:1:2:7039:7844:1
1:1:1:1:2:7844:30031:1
1:0:1:1:1:10097:10366:5
1:3:1:1:1:10366:10389:1
1:0:1:1:1:10389:10415:5
1:3:1:1:1:10415:1186112:1
1:13:1:2:1:29522:34145:1
1:0:1:1:2:30031:408501:5
1:16:1:2:2:31717:32188:1
1:16:1:2:2:32188:32715:1
1:0:1:2:2:32715:432738:5
1:0:1:2:1:34145:34380:5
1:13:1:2:1:34380:34402:1
...
```

Note, that resource model information has been defined and record traces contain the processor identifier where thread has been running. If no resources are used, a zero is filled in the processor identifier field of the record.

There are two NODEs with eight CPUs, one APPL. The APPL has two TASKs, the first one has ten threads an has been running in the NODE 1, the second one has also ten threads but has been running in the NODE 2.

Chapter 4

Configuration files

Paraver has two configuration files to configure some aspect of the environment. The user can customize things such as colors, name of the code colors and gradient colors, labels ...

4.1 Paraver Configuration File (.pcf)

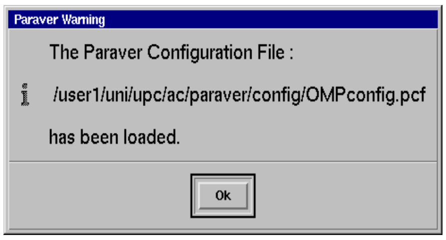
The main goal of the paraver configuration file is to offer to the user the possibility of configuring his/her own environment in which he/she has to work. This file is composed by different sections which give the possibility to define different aspects of the paraver environment such as colors, labels and some default options. The configuration is a plain text file.

4.1.1 When are Paraver Configuration Files loaded ?

A paraver configuration file can be loaded in different ways. Since they let to define some options like states (colors and labels) and user event labels they could be defined for a specific trace file, so each time a trace file is loaded the configuration file could be loaded. Also, if the user wants to use a defined environment for all the trace files we could load one when Paraver is launched.

Loading a paraver configuration file when Paraver is launched

In its initialization, Paraver checks for the environ variable PARAVER_CONFIG_FILE. If it exists, it must contain the global name of the configuration file that will be used. If it does not exists, the default configuration will be used.



The Paraver Configuration File :

/user1/uni/upc/ac/paraver/config/OMPconfig.pcf

has been loaded.

Figure 4.1: PARAVER_CONFIG_FILE environment variable

Example:

```
tcsh : setenv PARAVER_CONFIG_FILE   /user1/uni/upc/ac/paraver/config/OMPconfig.pcf
zsh  : export PARAVER_CONFIG_FILE = /user1/uni/upc/ac/paraver/config/OMPconfig.pcf
```

If the paraver configuration file is encountered, a message (figure 4.1) will be raised to show that it has been loaded. The defined event labels are copied to all the traces that will be loaded.

Loading a paraver configuration file when a trace file is loaded

The options (usually, colors and labels) defined in a paraver configuration file could be for a trace file. Paraver offers a way to load them when a trace file is loaded.

When loading a trace file, if there is a file named like the trace name but with the **.pcf** extension instead of **.prv/.map**, Paraver loads it automatically. For example, when loading the trace file called **NAS_Bt_OpenMP.prv**, if there is a paraver configuration file called **NAS_Bt_OpenMP.pcf** in the same directory it is automatically loaded.

4.1.2 Paraver Configuration File format.

This file is composed of eight sections but some of them could be ommitted (see below). The goal is override the default configuration in some aspects to make a most comfortable environment.

1. DEFAULT_OPTIONS section This section configure some general options of the paraver environment and **must be the first section** in the paraver configuration file. The section is composed by some tags and their value. Each of this tags and all this section could be omitted if the user does not want to override the Paraver default options.

   The format is :

```
DEFAULT_OPTIONS

option1     default_value_for_that_option
option2     default_value_for_that_option
...
```

   The options that can be changed in this section are (see figure 4.2) :

   - LEVEL : Set the default level for a visualizer window. His values could be : SYSTEM, NODE, CPU, THREAD, WORKLOAD, APPL or TASK.
   - UNITS : Set the default Paraver units. His values could be : MICROSEC, MILISEC, SEC or HOUR.
   - SPEED : Set the default value of the speed value. Remember, that speed is used when displaying a window to avoid X-terminal freezing (see the *Paraver Reference Manual* for a detailed description). The scrolling speed could be a value from 0 to 100 microseconds.
   - LOOK_BACK : Set the default percentage or traces to **look back** when semantic value couldn't be computed (see the *Paraver Reference Manual* for a detailed description). Its value could be a percentage from 1 to 100.
   - YMAX_SCALE : Set the default maximum Y-scale.
   - YMIN_SCALE : Set the default minimum Y-scale.
   - NUM_OF_STATES_COLORS : Change the number of code colors used in Paraver. By default, Paraver only works with 16 code colors (from code 0 to code 15). The user can change this number but have to define the selected number of states labels/colors in the STATES/STATES_COLOR sections.
   - DEF_CFG_DIRECTORY : Add the specified direcotry to the default directories when loading configuration files (see the *Paraver Reference Manual* for a detailed description).
   - DEF_PRV_DIRECTORY : Sets the default directory to load trace files.

   Also, there are four more options that could be changed which affect to the displaying windows (see figure 4.2) :

   - COLOR_MODE : Sets the default color mode for a displaying window. Values could be ENABLED (color mode is enabled) or DISABLED (color mode is disabled).

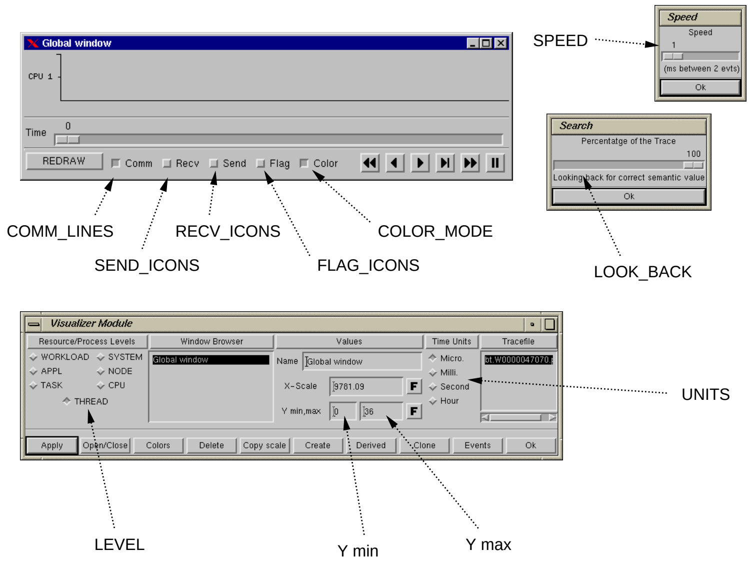


Figure 4.2: Default options section

- FLAG_ICONS : Sets the default value for flags. Values could be ENABLED (flags will be painted by default) or DISABLED (flags won't be painted).
- COMM_LINES : Sets the default value for communication lines. Values could be ENABLED (communication lines will be painted by default) or DISABLED (communication lines won't be painted).
- SEND_ICONS : Sets the default value for send icons. Values could be ENABLED (send icons will be painted by default) or DISABLED (send icons won't be painted).
- RECV_ICONS : Sets the default value for receive icons. Values could be ENABLED (receive icons will be painted by default) or DISABLED (receive icons won't be painted).

2. $\boxed{\text{DEFAULT\_SEMANTIC section}}$ This section configure which values will be the default in the Semantic Module. The user can select the default functions in object/compose levels and their default parameters.

   The format is :

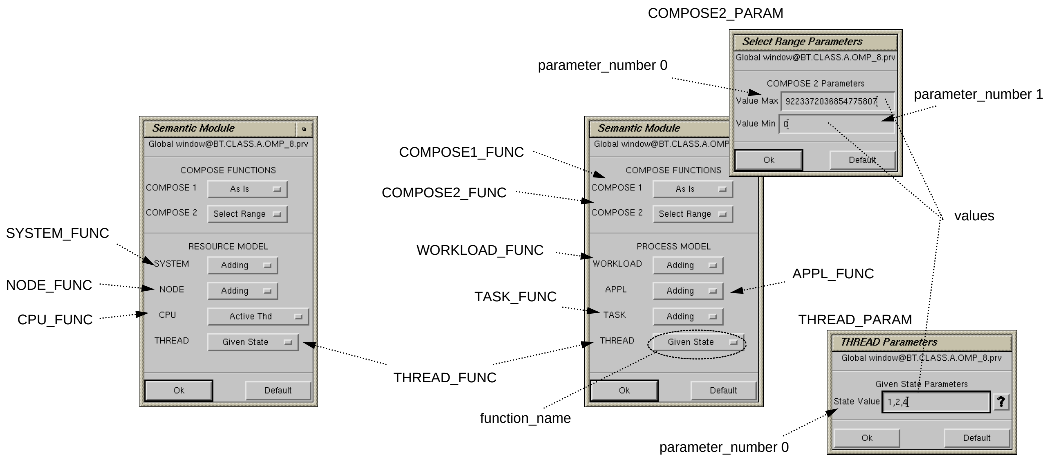
```
DEFAULT_SEMANTIC

tag_level           function_name
tag_level           function_name

tag_level_param     function_name      parameter_number     values
tag_level_param     function_name      parameter_number     values
tag_level_param     function_name      parameter_number     values
...
```

   The first group (tag *tag_level*) defines which function will be selected in each level as default; the second group defines the default parameters of a function (if the function has at least one parameter).

   The **tag_level** could be the tags : THREAD_FUNC, TASK_FUNC, APPL_FUNC, WORKLOAD_FUNC, CPU_FUNC, NODE_FUNC, SYSTEM_FUNC, COMPOSE1_FUNC or COMPOSE2_FUNC; where each



Global window@BT.CLASS.A.OMP_8.prv

COMPOSE FUNCTIONS
COMPOSE 1 As Is
COMPOSE 2 Select Range

RESOURCE MODEL
SYSTEM Adding
NODE Adding
CPU Active Thd
THREAD Given State

PROCESS MODEL
WORKLOAD Adding
APPL Adding
TASK Adding
THREAD Given State

COMPOSE 2 Parameters
Value Max 9223372036854775807
Value Min 0

Given State Parameters
State Value 1,2,4

Figure 4.3: Default semantic section

tag corresponds to a object/compose level. The **function_name** must be a case sensitive function name which exists in that level.

The **tag_level_param** tag could be : THREAD_PARAM, TASK_PARAM, APPL_PARAM, WORKLOAD_PARAM, CPU_PARAM, NODE_PARAM, SYSTEM_PARAM, COMPOSE2_PARAM or COMPOSE2_PARAM; where each tag marks the level of the function that will be changed. To define the default values for a parameter we have to tell the function_name, the parameter_number (because some functions could have more than one parameter like *Select Range* function in compose levels) and their values separated by comas. The first parameter number is 0, the second is 1, ... (see figure 4.3).

3. DEFAULT_FILTER section : Like the previous one, this section can be used to define the default functions and default values in the filter module.

   The format is :

```
DEFAULT_FILTER

tag       function_name      values
tag       function_name      values
tag       function_name      values
...
```

   The tag tells which function we are defining and could be : FROM_FUNC, TO_FUNC, TAG_FUNC, SIZE_FUNC, TYPE_FUNC or VALUE_FUNC (see figure 4.4). The function_name could be : All, =, !=, None, ¡ or ¿.

   The default values for each function have to be separated by comas.

4. DEFAULT_MICROSCOPE section : Like the previous ones, this section can be used to define the default functions and default values in the analyzer module. We can define the default function in each microscope row and the default values for each function and parameter.

   The format for this section is :

```
DEFAULT_MICROSCOPE

ROW1_FUNC    function_name
ROW2_FUNC    function_name
```

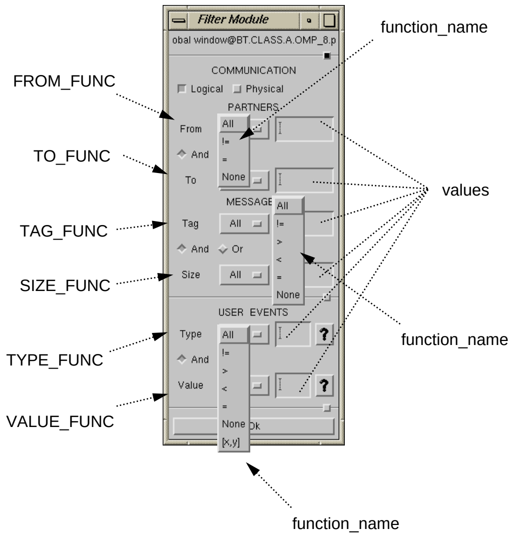
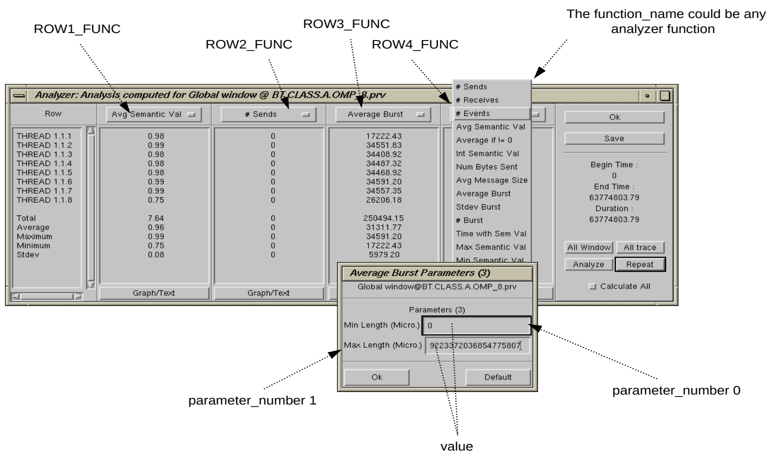


Figure 4.4: Default filter section

```
ROW3_FUNC    function_name
ROW4_FUNC    function_name
...
PARAM        function_name    parameter_number      value
PARAM        function_name    parameter_number      value
PARAM        function_name    parameter_number      value
...
```

Where in line ROWX_FUNC we are defining the default microscope function for each row and in each PARAM line we can define the default values for each parameter for a functions (see figure 4.6).



Row	Avg Semantic Val	# Sends	Average Burst
THREAD 1.1.1	0.98	0	17222.43
THREAD 1.1.2	0.99	0	34551.83
THREAD 1.1.3	0.98	0	34408.92
THREAD 1.1.4	0.98	0	34487.32
THREAD 1.1.5	0.98	0	34468.92
THREAD 1.1.6	0.99	0	34591.20
THREAD 1.1.7	0.99	0	34557.35
THREAD 1.1.8	0.75	0	26206.18
Total	7.64	0	250494.15
Average	0.96	0	31311.77
Maximum	0.99	0	34591.20
Minimum	0.75	0	17222.43
Stdev	0.08	0	5979.20

Begin Time : 0
End Time : 63774803.79
Duration : 63774803.79

Min Length (Micro.) 0
Max Length (Micro.) 9223372036854775807

Figure 4.5: Default microscope section

5. STATES section This section defines the label of the states that will be used. By default, Paraver has its own defined labels which give the meaning of each state (for example, state value 3 is defined as *Waiting a message*). The format is :

```
STATES
```

```
number_of_state              label
number_of_state              label
...
```

The tag STATES marks the beginning of this section. The section is composed by a list where the user has to specify the state value and its new label. The user can only specify the new desired states labels, for the rest will be used its default value or its new definition (only if a previous paraver configuration file has been loaded).

The number of state values that can be defined depends on the value defined in the **default options** section; if no new number has been defined the user only can define the labels from state value 0 to 15 (sixteen states).

6. $\boxed{\text{STATES\_COLOR section}}$ This section defines a new RGB color for each state. The format for this section is :

```
STATES_COLOR
number_of_state       {red,green,blue}
number_of_state       {red,green,blue}
...
```

The tag STATES_COLOR marks the beginning of this section. The section is composed by a list which where the user has to specify the new RGB color for each state that would be overridden. The numbers of the RGB must be values **0 and 255**, and define the red, green and blue basics colors.

The number of state values that can be defined depends on the value defined in the **default options** section; if no new number has been defined the user only can define the labels from state value 0 to 15 (sixteen states).

7. $\boxed{\text{GRADIENT\_COLOR section}}$ This section defines the RGB color for each gradient color. The format is :

```
GRADIENT_COLOR
gradient_color       {red,green,blue}
gradient_color       {red,green,blue}
...
```

The tag GRADIENT_COLOR marks the beginning of this section. The section is composed by a list which indicates the number of the gradient color and its RGB. The numbers of the RGB must be between **0 and 255**, and are the red, green and blue basics colors. Remember that Paraver works with 15 gradient colors so the number should be between interval **0...14**.

8. $\boxed{\text{GRADIENT\_NAMES section}}$ This section defines the label of the gradient colors that will be displayed within Paraver. The format is :

```
GRADIENT_NAME
gradient_color       label
gradient_color       label
...
```

The tag GRADIENT_NAME marks the beginning of this section. The section is composed by a list which indicates the number of the gradient colors and their label. Remember that Paraver works with 15 gradient colors so the number should be between interval **0...14**.

9. $\boxed{\text{Events Section}}$ This section is used to define the color and labels for the events. Each event type can have associated a gradient color which will be painted in the displaying windows. The format of this section is :

```
        EVENT_TYPE
        gradient_color        type          label
        gradient_color        type          label
        ...
     [  VALUES
        value     label
        value     label
        ...                ]
```

   And is composed by pairs of EVENT_TYPE and VALUES. You can define many pairs as you would. If you wouldn't define labels for the values of certain events types you can omit the tag VALUES for these event types.

   The EVENT_TYPE format is the gradient color which Paraver will use to paint the event, the type of the event and its label for the click utility. The VALUES format is the value of the event and its label for the click utility. The values affect only to the events type defined in the previous EVENT_TYPE tag.

Paraver distribution contains an example of paraver configuration file; also, in tutorial traces there is one paraver configuration files for each trace type. This paraver configuration files accepts lines with comments. A comment must start with the symbol #, when this symbol is encountered when processing the file the rest of the line will be ignored.

4.2 Names Configuration File (.row)

The names configuration file is used to change the default row names for a specific tracefile, this file can be generated within Paraver (see the *Paraver Reference Manual* for a detailed description). By default, each row label is composed by the level and its object identifiers (in task level are appl and task identifiers, in thread level are appl, task and thread identifiers).

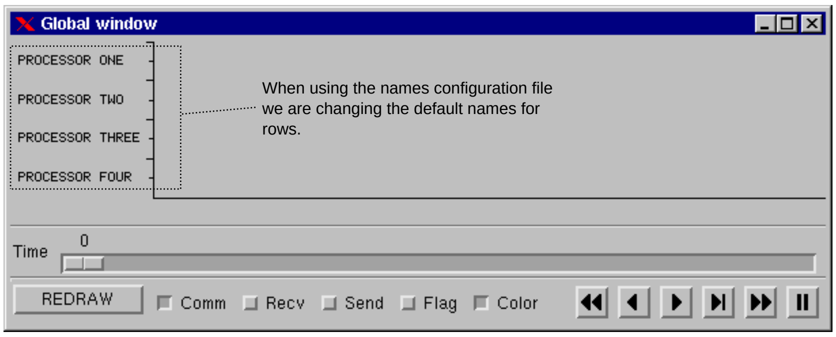


Figure 4.6: Default microscope section

The **names configuration file** is a plain text file where there are defined the new label for each tracefile object. There are four sections, one four each level, and all the sections must be in the file with the correct order.

All the sections have the same format and they are composed by a list of new names for the corresponding objects. The beginning of each section has the following tags :

```
LEVEL  "name of the level"   SIZE  "number of objects in this level"
```

The **name of the level** is the name of the type of objects (or name level), and could be SYSTEM, NODE, CPU, WORKLOAD, APPL, TASK and THREAD; and the **number of objects in this level** indicates how many objects are in this level. Note that this file is oriented to a tracefile and sometimes only it is useful for one tracefile or more than one trace files if they have the same number of objects in each level.

The order of the levels must be SYSTEM, NODE, CPU, WORKLOAD, APPL, TASK and THREAD. Also, this file accepts comments. A comment line must begin with the symbol # and the rest of the line will be ignored.

Example .- A names configuration file looks like :

```
#######################################################################
# NAMES CONFIGURATION FILE
########################################################################

########################################################################
# CPU LEVEL
# The format is :
#     LEVEL CPU  SIZE n
#     name of cpu 1
#     name of cpu 2
#     ...
#     name of cpu n
#######################################################################

LEVEL CPU         SIZE 8
PROCESSOR  0                # Object CPU 1.1
PROCESSOR  1                # Object CPU 1.2
PROCESSOR  2                # Object CPU 1.3
PROCESSOR  3                # Object CPU 1.4
PROCESSOR  4                # Object CPU 2.1
PROCESSOR  5                # Object CPU 2.2
PROCESSOR  6                # Object CPU 2.3
PROCESSOR  7                # Object CPU 2.4

#######################################################################
# APPL LEVEL
# The format is :
#     LEVEL APPL  SIZE n
#     name of appl 1
#     name of appl 2
#     ...
#     name of appl n
#######################################################################

LEVEL APPL       SIZE 2
MPI+OpenMP appl                  # Object APPL 1
OpenMP appl                      # Object APPL 2


#######################################################################
# TASK LEVEL
# The format is :
#     LEVEL TASK  SIZE n
#     name of task 1
#     name of task 2
#     ...
#     name of task n
#######################################################################

LEVEL TASK        SIZE 3
MPI Rank 0                          # Object TASK 1.1
MPI Rank 1                          # Object TASK 1.2
OpenMP task                         # Object TASK 2.1

#######################################################################
# THREAD LEVEL
# The format is :
#     LEVEL THREAD  SIZE n
#     name of thread 1
#     name of thread 2
#     ...
#     name of thread n
#######################################################################

LEVEL THREAD      SIZE 12
Master                      # Object THREAD 1.1.1
Slave                       # Object THREAD 1.1.2
Slave                       # Object THREAD 1.1.3
Slave                       # Object THREAD 1.1.4
Master                      # Object THREAD 1.2.1
Slave                       # Object THREAD 1.2.2
Slave                       # Object THREAD 1.2.3
Slave                       # Object THREAD 1.2.4
Master                      # Object THREAD 2.1.1
Slave                       # Object THREAD 2.1.2
Slave                       # Object THREAD 2.1.3
Slave                       # Object THREAD 2.1.4

#######################################################################
# NODE LEVEL
# The format is :
#     LEVEL NODE  SIZE n
#     name of node 1
#     name of node 2
#     ...
#     name of node n
#######################################################################

LEVEL NODE        SIZE 2
Node 0                             # Object NODE 1
Node 1                             # Object NODE 2

#######################################################################
# SYSTEM LEVEL
# The format is :
#     LEVEL SYSTEM  SIZE n
#     name of the level
#######################################################################
```

```
LEVEL SYSTEM        SIZE 1
4-way SMPs                         # Object SYSTEM

######################################################################
# WORKLOAD LEVEL
# The format is :
#     LEVEL WORKLOAD  SIZE n
#     name of the level
######################################################################

LEVEL WORKLOAD        SIZE 1
Workload (2 applications)          # Object WORKLOAD
```