


---

This is the **accepted version** of the book part:

Espinosa, Antonio; Margalef, Tomàs; Luque, Emilio. «Automatic performance evaluation of parallel programs». A: Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing - PDP '98. 1998, p. 43-49. Institute of Electrical and Electronics Engineers (IEEE). DOI 10.1109/EM-PDP.1998.647178

---

This version is available at <https://ddd.uab.cat/record/288088>

under the terms of the  <sup>IN</sup> COPYRIGHT license

# Automatic Performance Evaluation of Parallel Programs

Antonio Espinosa, Tomàs Margalef and Emilio Luque  
Computer Science Department  
Universitat Autònoma de Barcelona  
08193 - Bellaterra (Barcelona). SPAIN  
{a.espinosa, t.margalef, iinfid} @cc.uab.es  
Phone: + 34 3 581 19 90

## Abstract

*Traditional parallel programming forces the programmer, apart from designing the application, to analyse the performance of this recently built application. This difficult task of testing the behaviour of the program can be avoided with the use of an automatic performance analysis tool. Users are released from having to understand the enormous amount of performance information obtained from the execution of a program.*

*The automatic analysis bases its work on the use of a pre-defined list of logical rules of production of performance problems. These rules form the “knowledge base” of the tool. When the tool analyses an application, it looks for the occurrence of an element in the list of performance problems recorded in the “knowledge base”. When one of the problems is found ( a “match” in the list), the tool analyses the cause of the performance problem and builds a recommendation to the user to direct the possible modifications the code of the application.*

## 1.- Introduction.

Performance analysis of parallel programs has traditionally been addressed as the last of the steps to design an application. Programmers of applications, with the help of performance visualisation tools like Paragraph [Heath 91] and Pablo [Reed 93], have to understand the behaviour of the applications running on the target machines.

Those kind of tools present the performance of an application as a collection of graphical views. The views summarise the behaviour of certain aspects of the program during an execution interval. Users must look for sensitive views of the execution, redefining them until the behaviour of the application is completely understood. The disadvantages of these tools remain in the difficulty of understanding the enormous amount of performance data generated for an application and the difficulty of relating the performance views with the corresponding part of the application.

Other approaches have considered the analysis of the applications from a certain parallel programming model [Crovella 94] [Meira 95]. The analysis consists of building an abstract expression of the application according to the general model. From that point on, this expression represents the behavioural characteristics of the application. The analysis predicts certain aspects of the program future behaviour when considering the analysis of the abstract expression. The main problem of this approach is that the abstract representation of the program may hide relevant aspects of the program performance, which are not considered in the programming model.

When parallel application programmers want to improve the performance of their applications, the usual procedure is, after generating a trace file from the execution of the application, use a visualization tool to search for execution problems. With some experience in understanding the performance views, programmers are able to locate the most important problems occurred at the execution of the application. After locating the problems in the views, the next step is to understand the causes of those performance problems found.

After studying some parallel programming models and the performance of programs under those models, we have developed a list of performance problems that can be found at the execution of an application. With the help of this list, problems can be analysed to derive conclusions about their causes and possible changes in the code of the application to solve them.

In this paper, we describe an automatic performance analysis tool which, using a monitorization scheme, pretends to release users from analysing the overwhelming quantity of performance information generated for the execution of an application. The tool automatically selects from the available information of the program (trace file, application code,...) whatever it finds relevant to analyse and improve the behaviour of the application.

The automatic tool analyses the inefficiency intervals found in the trace files. These inefficiency intervals represent time periods when the application is not using all the capabilities of the parallel system. If the application could extract all the potential of the machine during the execution time it surely will be completed in a shorter time, setting its performance to the maximum expression.

The tool divides this work of analysing the execution of the application in some phases:

- A general analysis phase, when the tool tries to locate general behavioural problems in the execution. The tool scans the trace files classifying the execution of the application depending on the efficiency of the system during the execution time.
- If the general efficiency of the system does not meet the requirements of the design (for example, execution is too slow) the tool starts looking for the most important performance problems of the application. These problems will be stored in a list ordered by execution importance.
- The most important performance problems are selected to be analysed in detail. The tool focuses its work in finding the causes of these problems, using an internal list of causes of performance problems ( the “knowledge base” of the tool). This analysis process consists of finding which causes “match” with a specific problem.

When all causes of a problem are found, the tool builds a report of suggestions on how to overcome the problems found and presents it to the user/programmer of applications.

There are other automatic performance analysers, like Paradyn [Waheed 96], that analyse the performance at execution time, therefore they must focus its work in minimising the overhead of introducing monitoring control of the system. For that reason, Paradyn possibly misses problems which fall out from the current instrumentation bounds.

In section 2, we describe the list of performance problems (“knowledge base” of the tool) to be used as a reference pattern at the analysis of the application. In section 3, we introduce the implementation of the analysis tool. Finally, in section 4, we describe the future work to be done on the tool development.

## 2.- Classification of the performance problems.

The list of general problems found at the analysis of parallel programming models, added to the experimentation on PVM parallel programs, has derived in a specification of performance problems and their causes.

The analysis of the inefficiency intervals, defined at the introduction, makes an initial differentiation between two problems:

- During the inefficiency interval there are idle processors and there are not tasks ready to execute, neither time-sharing executing tasks at one of those processors. The analysis must find out the reasons of this *lack of ready tasks*. The aim of the analysis is to fill those inefficiency intervals. To do so, it must find which changes should be applied to the program to activate some tasks in those intervals.
- *Mapping problem*: there are idle processors during the inefficiency interval and, at the same time, there are ready tasks in other processors. The analysis must consider the states of the execution trying to modify the current task-processor mapping. This process involves the consideration of changes in the communication delays when a task is mapped on a different processor.

From these two initial situations the analysis must try to make a deeper analysis of which are the actual causes of this two performance problems, providing some suggestions to the programmer. However, these two situations are not mutually exclusive, i.e. , it is possible to find a poor performance interval with some ready tasks in other processors and, when trying to determine which task should be mapped in the idle processor, the analysis concludes that it is not possible to move any task because of the communication constraints. Then, other branches of the analysis should be considered.

In figure 1, we classify the performance problems and its causes. These problems and its causes are going to be explained in detail in the following points.

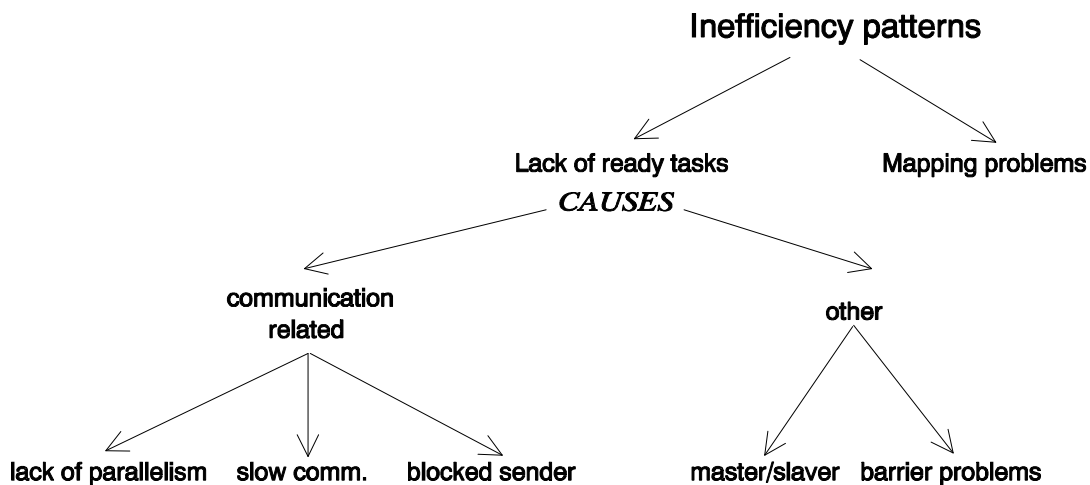


Figure 1: classification of performance problems detected by the analysis and its causes.

## 2.1 Lack of ready tasks

When the analysis finds that there are no ready tasks in the system to be executed on some available processors it must find which tasks could be activated to fill that void. The first tasks analysed are those which start executing just after the inefficiency intervals. The analysis focuses on finding why those tasks do not execute previously and what should be modified in order to activate them before.

This process of finding the reasons of what prevents the involved tasks to execute earlier is equivalent to explain the causes that provoke the inefficiency interval. The analysis uses a list of possible causes of inefficiency intervals, the “knowledge base” derived from the analysis of problems of the parallel programs, to analyse why a task does not execute before.

The most common cause of an inefficiency interval is that a task is blocked involved in a communication (and there are no other ready tasks in the same processor). In our case the blockings are produced in the receive message operations. From here, we have divided the list of inefficiency causes in two sections related or not with communications.

1.- The first part of the list is composed by communication-related causes:

a) *Lack of parallelism:*

The inefficiency interval is produced by the blocking of some tasks which must wait for a message from another task. This task must generate data for all the rest, deriving a serialisation of the computation. While this task executes in a certain processor the other tasks wait for the message, leaving their processors idle.

b) *Slow communication:*

Task is blocked waiting for waiting for the reception of a message which has already been sent. The cause of this slow movement of the message could be the (long) size of the message itself or a contention in the network between the processors.

c) *Blocked Sender:*

The blocking of the communications is produced by the sender task. The communication is blocked because the sender, instead of sending the required message, is itself receiving a message from other tasks. In this case, it is necessary to analyse the application to determine if there exists a data dependence between all communication partners. That is to say, the analysis must test the independence of the tasks in communication. If they are tested independent, it should be possible to modify the order of the sentences to avoid this overlap.

d) *Multiple output problem:*

A task is blocked waiting for a message and the sender has not started the transmission yet because it is sending other messages to other tasks.

2.- Moreover, there are other causes, not directly involving communication, which also provoke inefficiency intervals:

e) *Problems at a barrier:*

When there is a barrier in the application and the tasks reach it with too different times, the first task which reaches the barrier remains blocked until the last one arrives.

f) *Problematic Master/Slave relationships.*

Under this collaboration scheme the master tasks may need a minimum number of slaves to assure a good performance behaviour

In the following points, the causes of a lack of ready tasks problem are analysed in detail, describing the kind of solutions that the analysis is likely to suggest for each cause.

### 2.1.1 Lack of parallelism.

A lack of ready tasks problem may be caused by a not scalable application design which, during an execution interval, has not enough ready-to-execute tasks for all processors of the system. When analysing the tasks which execute after the inefficiency interval, we find that they cannot execute previously (and therefore, fill the inefficiency interval) because of their dependencies on the currently executing tasks in the program (see figure 2, last task on cpu1).

In these situations, the main recommendation is to redesign the application in order to distribute the calculations performed during the inefficiency interval among all processors. In this way, we pretend to increase the scalability of the program.

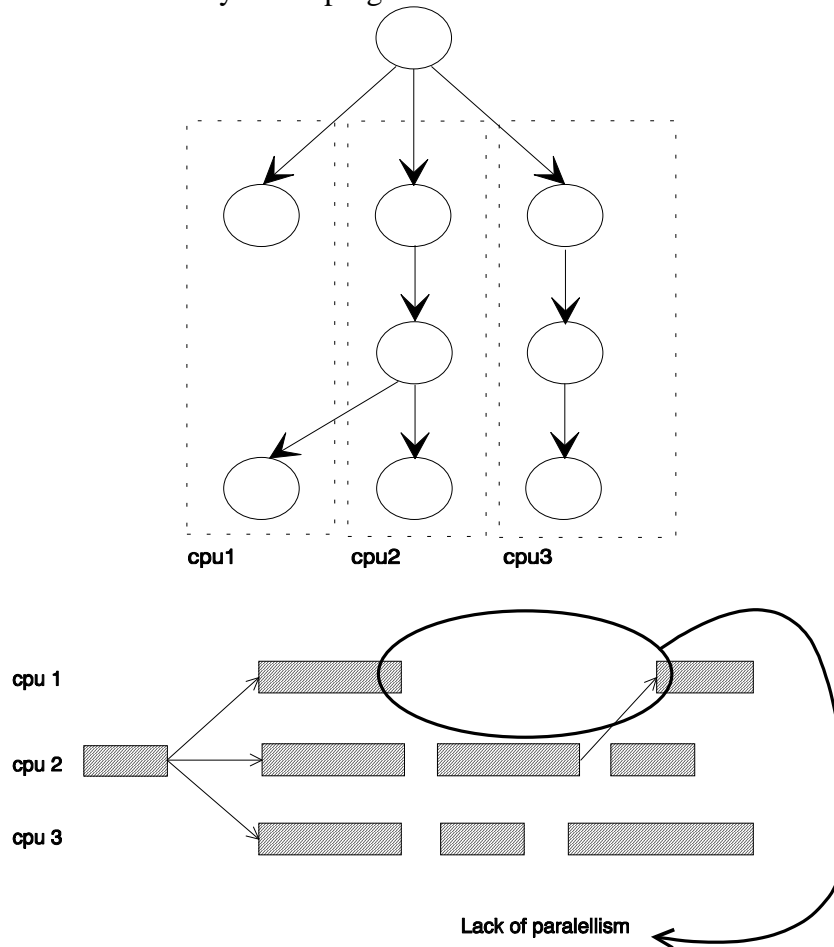


Figure 2: Lack of parallelism in the second level of the design.

### 2.1.2. Slow communications

An interval of inefficiency may be caused when the communication between two tasks becomes so slow that the execution of the application is serialised.

In the experimentations, the analyses were carried out on applications under PVM programming model. A task which wants to receive a message makes a call to the receive message primitive

and starts waiting for the message to come. This blocking of the receiver tasks is one of the most common sources of inefficiency intervals.

When analysing those intervals, it rebuilds the communication history of the tasks involved. This problem of slow communications will be detected when the inefficiency interval equals to the time that the message has spent moving along the interconnection network of the machine.

Suggestions for this problem consider the state of the network (mainly the occupation grade) and the size of the messages transmitted, this information must be found out from the trace files actually used. To be able to analyse this problem it would be necessary to define new monitorization processes to recover this kind of information of the system. Solutions contemplate the remapping of the communication partner to a nearer processor and, if the target machine allows it, redirect the message through less loaded parts of the network.

### 2.1.3. Blocked sender.

In this case, the inefficiency interval is, again, an interval of blocked communication. This time, when analysing the history of the transference we discover that the sender of the message is itself waiting for a message from a third task. At least we have three tasks involved, so this relationship must be studied in detail.

This three (or more) way relationship can be a coincidence in time, and then the tasks execution order can be modified to avoid this overlap. On the contrary, the relationship could also represent a data dependence between a group of tasks, which could be affecting negatively the performance of the application. From this point on, the independence of the tasks involved in the communications must be tested.

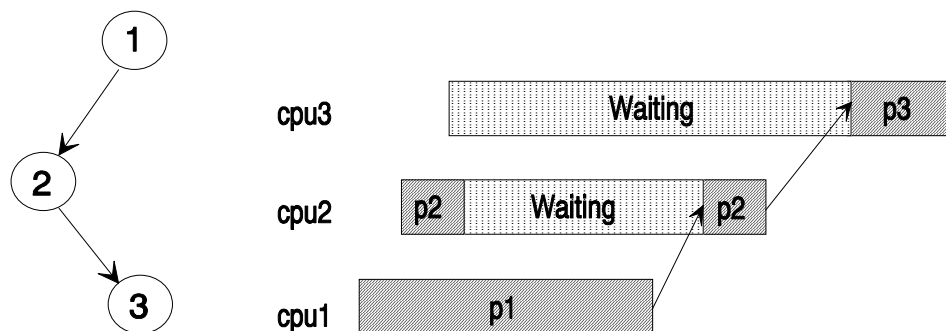


Figure 3: Task 3 needs some data from task 2, while task 2 waits for the reception of some data from task 1.

In the application represented by the graph of figure 3, we detect a communication relationship of three tasks. The analysis must differentiate between a real dependence or a coincidence in time of the communications of the tasks. A data dependence will be found when task 3 blocks for reception of a data item sent from task 2. At the same time, task 2 is blocked to receive a message from task 1. If these two messages are related somehow (for example, one message is a part of the other) the analysis will notify this situation and suggest how to distribute the data in a better way.

To consider all data movements between tasks in a group, the analysis builds a table of transferences attending the sender, the receiver and the timing of each transference. This table is built for all communications in the execution. The table is completed with the data-structure related with the transferred messages which must be found at the code of the application. Once

the table of transferences is built, the analysis addresses the following questions, building an answer for each:

- Which data transference generates the problem? (task which blocks longer)
- Is there any owner task of the data?
- Does this data dependence appear other times in the execution of the application? (with the same tasks)
- Are there any other dependencies of other processes with the same data item?

In this way all dependencies of three or more tasks are considered, trying to bring to the light possibly unadvertised relationships between them.

#### 2.1.4. Multiple output problem.

Another kind of communication blocking is produced when a process has a list of son tasks. If, as shown in figure 4, all of them block to receive a message from the father, we may be in front of a serious blocking problem. Figure 4 represents a typical multiple output construction. As long as S task sends all messages to the receivers in parallel, the structure will not represent a problem.

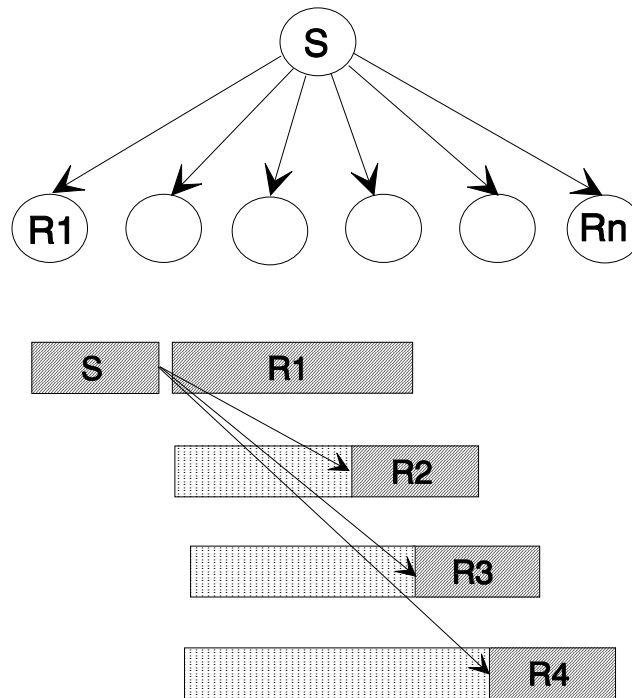


Figure 4: Potentially problematic communication scheme when considering blocking receive primitives.

But, if all son tasks start the blocking reception of a message at the same time, and they receive the messages one by one, these messages will unblock the receivers in a serial manner as the messages arrive. The last task to receive a message is the task which waits longest.

Everytime that a problematic blocking of a task waiting for a communication is detected and, at the same time, the sender task is sending messages to other tasks in the application, we will have the same problem. We will call it multiple output problem.

When the analysis verifies the existence of a multiple output problem, it will suggest the possibility of using a broadcast of the messages or, if the data messages are different for each son task, use a scatter operation to distribute the disjoint messages.



### *2.1.5. Problems at a barrier.*

Barrier primitive may be called from any task within a group of tasks which collaborate to solve a problem. The barrier primitive blocks all tasks which call it until a fixed number of tasks reach the barrier. When a task arrives at the barrier primitive call considerably later than the other tasks, all of them must wait for the last, therefore blocking the execution of the application.

The analysis focuses its work on finding the cause of the last arrivals at the barrier. Those tasks which arrive latest are analyzed in detail to find any blockings (or unbalanced computation) along the execution. These blockings are explained to the user as causes of the barrier problem.

### *2.1.6. Problematic master/slave relationships.*

When considering the analysis of the application, i.e. finding the causes of performance problems produced in the execution, a special feature is the ability to look for the execution of master-slave collaboration schemes in the trace files. In these schemes, a master task generates a data item and sends it to a slave task to process the data. Immediately after the master sends the data item to the slave, it starts computing a new data item to send. From this operational rules, it is clear that if the master task generates data items frequently, the number of the slaves should be high enough to cope those necessities. From here, the analysis tries to find out how many slaves does the master need, suggesting the creation of new slaves or the partition of the currently executing slaves to create more simpler slaves.

To detect the needs of the master, the analysis locates the master (or masters) and the slaves. Then, it considers:

- The blocking times of the master tasks. If it is too high, it will recommend the creation of new slaves. The less the master has to wait, the faster the application.
- The waiting for a message (which is already in transmission) times of the slave tasks. If the messages move too slowly (slaves wait for an important amount of time while the message moves along the network), the recommendation may be to change the mapping to bring the slaves near to the master. And also, suggest a reduction in the number of the slaves to avoid the contention of the messages in the net (output messages queue in master tasks).

## **2.2. Mapping problems**

When, considering an inefficiency interval, there are some ready tasks in some busy processors the analysis tries to reassign those ready tasks in order to fill the inefficiency intervals.

Firstly, all tasks which are suitable to be reassigned are considered. Secondly, all these relationships are analysed to generate a list of task candidates to be assigned to another processor. And thirdly, all the new task-processor tuples are analysed concerning the improvement of the application behaviour when the mapping change is produced.

The purpose of the analysis, when solving this problem, is to redistribute the ready tasks in the inefficiency intervals. For the changes in the task-processor mapping which improve the performance of the application, a suggestion to the programmer of the application is built.

In figure 5, CPU1 and CPU3 are idle waiting for a message from task p6, while tasks p3, p4, p5 and p6 are waiting for CPU2 to be executed. These ready tasks could be reassigned to CPU1 and/or CPU3 in order to complete p6 as soon as possible.

The analysis must also consider the possibility of remapping the same task several times along the execution. In this case, the dynamic mapping is considered as a possibility to maximize performance.

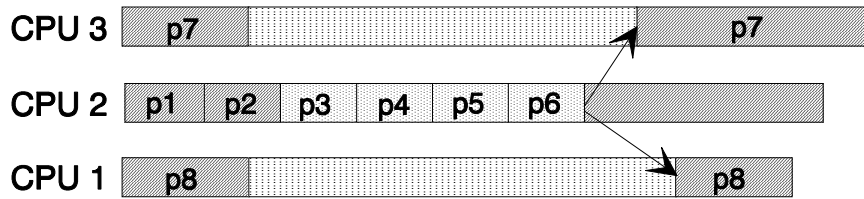


Figure 5: Mapping problem.

However, it is not always possible to modify the task-processor assignment due to communication constraints (if a task is changed to another processor, its distances to its communicating partners may have grown). In this case, the analysis is not able to provide any suggestion on this branch of the research tree and must try other branches (lack of ready tasks).

### 3.- Automatic Performance Analysis Tool implementation.

Our automatic performance analysis tool currently analyses applications under the PVM programming model. The tool analyses the trace files obtained by the use of Tape/PVM monitoring libraries [Maillet 95].

The most important work of the tool is, once a performance problem is found important, to analyse the causes that have provoked the problem. For that matter, the tool has a rule-based list of causes of performance problems. These rules are “matched” with the problem specification, obtaining a list of possible causes of the analysed problem. Currently, the list of causes is composed by some problems found during the experimentation with PVM parallel programs. The list specification is open to include new refinements of the problems and other parallel programming models.

The main operations of the tool are programmed with Perl. This language provides facilities to deal with large trace files and also is capable to read application sources written in any language. These two facilities are quite important when generating reports of suggestions to the user.

### 4.- Conclusions and future work.

The analysis steps are completely transparent for the users/programmers of applications, users bring its PVM application to the tool and receive a list of recommendations to apply to the application code. In this way, users are free from the tedious analysis of figures and statistical data which represent the behaviour of the program.

Nonetheless, when a problem is found and analysed, other new problems could appear when applying the suggested changes to the code. Programmers must be aware that focusing on solving the problem does not imply an automatic improve in the behaviour of the code. All changes in the code produce side effects on the behaviour of the rest of the application.

The list of suggestions should be understood a list of hints on understanding some aspects of the program. Their meaning is to guide the user in changing the appropriate parts of the application.

Future work on the automatic tool will refine the causes of performance problems and extend the tool usage to other parallel programming models.

Other important features of the tool to be improved are the information searches. The tool must carry out two kind of examinations: the search for inefficiency intervals and the “matching” between the causes of problems and the problems found. These examinations must be optimised to deal with very large trace files, the usual case in analysing parallel program executions.

## 5.- References

[Crovella 94]: Mark E. Crovella and Thomas J. LeBlanc. “The search for Lost Cycles: A New approach to parallel performance evaluation”. TR479. The University of Rochester, Computer Science Department, Rochester, New York, December 1994.

[Heath 91]: Michael T. Heath, Jenniffer A. Etheridge: "Visualizing the performance of parallel programs". IEEE Computer, November 1995, vol. 28, p. 21-28 .

[Maillet 95]: Maillet, Eric: ”TAPE/PVM an efficient performance monitor for PVM applications-user guide”, LMC-IMAG Grenoble, France. June 1995.

[Meira 95]: Wagner Meira Jr. “Modelling performance of parallel programs”. TR859. Computer Science Department, University of Rochester, June 1995.

[Reed 93]: D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz and L. F .Tavera: "Scalable Performance Analysis: The Pablo Performance Analysis Environment". Proceedings of Scalable Parallel Libraries Conference. IEEE Computer Society, 1993.

[Waheed 96]: Abdul Waheed, Diane T. Rover, Jeffrey Hollingsworth. “Modeling, evaluation and testing of paradyn instrumentation system” Proceedings of the Supercomputing Conference 1996.