Universitat Autònoma de Barcelona

Extending the Usability of a Dynamic Tuning Environment

Departament d'Arquitectura de Computadors

i Sistemes Operatius

Thesis submitted by **Paola Guadalupe Caymes Scutari**
in fulfillment of the requirements for the degree of Doctor per la Universitat Autònoma de Barcelona

Barcelona (Spain), July 2007

Universitat Autònoma de Barcelona

Extending the Usability of a Dynamic Tuning Environment

Departament d'Arquitectura de Computadors

i Sistemes Operatius

Thesis submitted by **Paola Guadalupe Caymes Scutari**
in fulfillment of the requirements for the degree of Doctor per la Universitat Autònoma de Barcelona

Barcelona (Spain), July 2007

Extending the Usability of a Dynamic Tuning Environment

Thesis submitted by **Paola Guadalupe Caymes Scutari** in fulfillment of the requirements for the degree of Doctor per la Universitat Autònoma de Barcelona. This work has been developed in the Computer Architecture and Operating Systems Department of the Universitat Autònoma de Barcelona, inside the program of Phd. in Computer Sciences, Option 'A': "Computer Architecture and Parallel Processing" and was advised by Dra. Anna Morajko and Dr. Tomàs M. Margalef Burrull,

Bellaterra, July 2007

Thesis Advisor

Anna B. Morajko

Tomàs M. Margalef Burrull

Acknowledgments

FIRST of all, I want to thank all the staff of the Department. The help given by them has always been very useful. In particular, I would like to thank my colleagues of the performance group, who participated in some manner in this work. I want to express my special thanks to Anna Morajko and Tomàs Margalef for their guidance, opinions and corrections and to Emilio Luque for his help and opinions. Thanks to the technical staff, Dani and Jordi for their impeccable technical support, and to Gemma and Oriol for the conversations and their helpful administrative support.

I also want to express my gratitude to the professor John Gurd and his group of collaborators for welcoming me and for their hospitality during my stay in Manchester. The experience in their research centre has been very interesting and useful. My thanks to Mikel Lujan for his comradeship, and the advices provided for my arrival.

These years in Barcelona have been a very rich experience of life. I have met a lot of people from very varied countries and cultures. Thanks for the talks and the shared moments, thanks to everyone for demonstrating that beyond the race, the creed and the religion, everybody wants a world without wars. Some of them became very good friends of mine; thanks to them and to my lifelong friends for their constant support and affection.

Very special thanks to all my dear family, for their love, their support and the comprehension they give me every day. Thanks to them for encouraging me in every step of my life, either the personal or the professional. Thanks!

My beloved Germán! thank you for your infinite love and your unconditional support and patience along the endless hours dedicated to this work. Thank you for the embrace and the shoulder in every moment I got disheartened, and for your wonderful smile every time I achieved some successful progress. Thank you for your opinions and help, and for being an extraordinary colleague, friend and husband. Thanks, thanks, thanks!

Finally, I want to dedicate this work to everybody who is in my heart.

¡Muchas gracias a todos!

To my grandparents

Contents

List of Figures **v**

Preface **ix**

1 Introduction **1**

1.1 Parallel Programming aspects 2

1.2 Performance of parallel applications 5

1.3 Performance Analysis and Tuning 9

1.3.1 Classical performance analysis 11

1.3.2 Automatic performance analysis 13

1.3.3 Dynamic performance analysis 14

1.3.4 Dynamic performance tuning 16

1.4 Thesis Contribution 19

1.4.1 Work organization 21

2 Monitoring, Analysis and Tuning Environment **23**

2.1 General view of MATE 24

2.1.1 Dynamic tuning 24

2.1.2 Additional characteristics 25

2.1.3 Dynamic Instrumentation: DynInst 27

2.2 MATE 30

2.2.1 Architecture 31

2.2.2 Application Controller - AC 33

2.2.3 Dynamic Monitoring Library - DMLib 35

2.2.4 Analyzer 36

2.3 MATE as a development and tuning environment 39

3 Scalability of Analyzer **43**
3.0.1 Centralized Analysis Approach 44
3.0.2 Scalability of MATE: Motivation 45
3.0.3 Different Approaches to support Scalability 48
3.1 Distributed-Hierarchical Approach 51
3.1.1 Global Analyzer 53
3.1.2 Collector-Preprocessor - CP 54
3.1.3 Validation of hierarchical-distributed approach 56
3.1.4 How to decide the number of CPs? 62
3.2 Overhead caused by MATE 64
3.2.1 Intrusion 65
3.2.2 Additional Resources 70

4 Automatic Development of Tunlets **73**
4.1 Introduction 74
4.2 Abstractions and Terminology 76
4.3 Methodology 81
4.3.1 Providing a performance model 82
4.3.2 Understanding the performance model 82
4.3.3 Interpreting the performance model 83
4.3.4 Identifying the actors in the application 85
4.4 Simple Example 85
4.5 Tunlet Specification Language 91
4.5.1 Components and Sections of the specification 93
4.5.2 Symbols 98
4.5.3 Syntax 104
4.5.4 Grammar 106
4.5.5 Semantics 111
4.5.6 Deducing CPs logic from the specification 119
4.6 Tunlet Generator Implementation 124
4.7 What does the User need to know? 125
4.7.1 Developing the parallel application 128
4.7.2 Defining the tunlet abstractions 128
4.7.3 Writing the specification of the tunlet 128

4.7.4 Special considerations and constraints 131
4.7.5 Generating the tunlet 133
4.7.6 Executing the application under MATE 133

5 Use Cases **135**
5.1 Optimal Number of Workers 136
5.1.1 Providing a Performance Model 136
5.1.2 Interpreting the Performance Model 138
5.1.3 Identifying the Actors 141
5.1.4 Tunlet Specification 143
5.1.5 Generated Tunlet 149
5.1.6 Experiments 150
5.2 Load Balancing 151
5.2.1 Providing a Performance Model 152
5.2.2 Interpreting the Performance Model 154
5.2.3 Identifying the Actors 157
5.2.4 Tunlet Specification 158
5.2.5 Generated Tunlet 160
5.2.6 Experiments 161

6 Conclusions **163**
6.1 Conclusions 163
6.2 Open Lines 168

A Tunlet Specification Language: Syntax Directed Definition **171**

B Tunlet Specification: Optimal Number of Workers **185**

C Tunlet Specification: Load Balancing **199**

Glossary **215**

Bibliography **219**

List of Figures

1 Two eras of computing x

1.1 Parallelizing a sequential problem. 7
1.2 Classical performance analysis approach 11
1.3 Automatic performance analysis approach 13
1.4 Dynamic performance analysis approach 15
1.5 Dynamic performance tuning 16

2.1 DynInst API abstractions 29
2.2 Operation of MATE 31
2.3 Architecture of MATE 32
2.4 Dynamic Tuning API class diagram 37
2.5 Elements of a tunlet 39
2.6 View of the Master/Worker parallel algorithm 40
2.7 Class diagram of the Master/Worker Framework 40

3.1 Analyzer interacting with the rest of the environment 44
3.2 Analyzer operation 46
3.3 What happens when the Analyzer is overloaded 47
3.4 Distributed-Hierarchical Collecting-Preprocessing Approach 52
3.5 Algorithm followed by Global Analyzer 54
3.6 Algorithm followed by Collector-Preprocessor 55
3.7 Sequence of events managed by the Centralized Analyzer 58
3.8 Events managed by the Distributed/Hierarchical Analyzer 59
3.9 Trace of the Analyzer for a given iteration 60
3.10 Comparison among real iteration time and the obtained for each approach . . 61
3.11 Comparison among times obtained for each approach 62

4.1 Automatic development of a tunlet from a specification 76
4.2 General functioning of MATE 77
4.3 Role of a tunlet in MATE 78
4.4 Relation between a performance model and a tunlet 80
4.5 Abstractions in the application 81
4.6 Interrelation among application, performance model and tunlet 82
4.7 Two simple and equivalent programs 84
4.8 Example of a simple parallel application 86
4.9 General view of a parallel application and the iteration time calculation . . . 87
4.10 Mathematical model of the iteration time calculation 89
4.11 General syntax of specifications 104
4.12 Properties to define a variable or value 105
4.13 Properties to define actors and events 105
4.14 Properties to define iteration information, tuning points and performance functions and parameters 106
4.15 Translation Scheme for transforming user entities in MATE entities 114
4.16 Data structure to store the attributes of a particular actor 120
4.17 Specification of lastWorker attribute of master actor 121
4.18 Aux data structure in subordinated Analyzers 121
4.19 Phases in the generation of a tunlet from a specification 124
4.20 General operation of MATE 126
4.21 Template to specify a tunlet 134

5.1 Execution times of NBody when considering the number of workers 151
5.2 Execution times of NBody when considering load balancing 162

Preface

IN the last years, computing performance demand has been in increase. This necessity appeared specially in different scientific areas that have to solve complex problems. Thus, Biology, Physics and Chemistry are becoming the main producers and users of applications with high performance computing requirements. There are many applications that differ from the functional point of view, such as the determining of the human genome, the simulation of the universe, nature models study, etc. However, in general the data set size and the complexity of the operations over them require the use of very powerful systems in order to solve the problem as fast as possible and using the resources in an efficient way.

Thus, the increasing necessity for high performance systems/computing has been directing the attention of the scientific field towards the parallel/distributed paradigm because of grand challenge applications very often need more computing power than a sequential computer can provide. In such sequential systems, the improvement of the operating speed of processors and other components is constrained by the speed of light, thermodynamic laws, and the high financial costs for processor fabrication. A viable and cost-effective alternative solution is to connect multiple processors together and coordinate their computational power. The resulting systems are popularly known as *parallel computers*, and they allow for the sharing of a computational task among multiple processors [8]. This general definition includes different kinds of parallel computers, such as one machine with thousands of processors or a set of workstations (a.k.a. *nodes*) connected through a local area network. Either of these configurations should provide a significant increase in the performance when compared to a uniprocessor. The general and basic idea is that n processors or nodes should provide a computational speed n times faster than a simple node, i.e., the problem should be solved in an interval of $1/n$ of the time [18]. Clearly, the advantages of using parallel systems constitute an ideal situation which in practice is not always true. However, even though parallel systems have some limitations in execution time, those limits are upper than the uniprocessors ones.

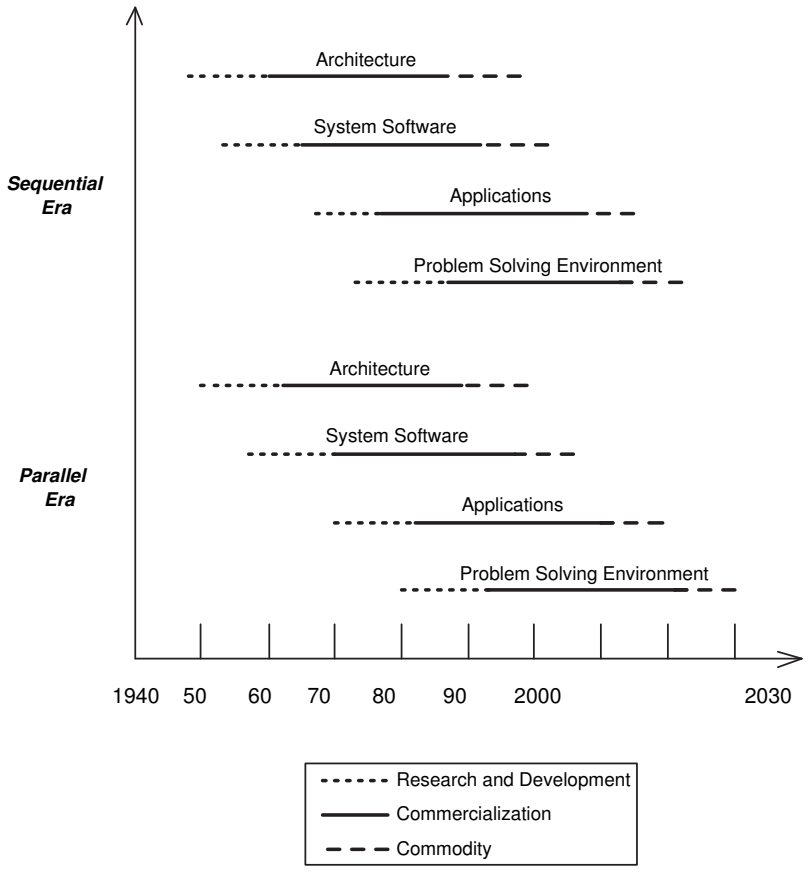

Figure 1: Two eras of computing

Figure 1 shows how an assortment of new architectures, compilers and operating systems have been developed through the years to enable the use of parallel computing, and cover the necessity of improving the computational performance [8]. All of them have been evolving as the sequential paradigm was finding some constraints to cover certain expectations. Programmers have to face a series of difficulties to reach the best performance of their applications. The development of parallel applications has to follow a specific manner to allow for their execution in a parallel system. In addition, once the application has been implemented, it has to be systematically tested from the functional point of view in order to guarantee its correctness. Following that, the application has to be adjusted to ensure that no bottlenecks exist in the execution, and in consequence that fulfils the aim of providing a better

performance. Thus, the performance of the parallel applications is a key aspect due to the difference between expected and real performances should not be significant. The optimization process, so-called *tuning process*, is -as its name indicates- the process followed in order to adapt and improve the behaviour of the applications by modifying their critical parameters.

The tuning process includes several and successive phases. Firstly, during a *monitoring phase* the information about the behaviour of the application is captured. Next, the information is *analyzed*, by looking for bottlenecks, deducing their causes and trying to determine the adequate actions to eliminate them. Finally, appropriate changes have to be applied to the code to solve the problems and improve the performance. As a consequence, the developers are forced to know very well the application, the different involved software layers and the behaviour of the distributed system. All these issues make difficult and costly the performance tuning process, specially for non-expert users, due to a high degree of expertise is required in order to significantly improve the behaviour of the application. Fortunately, through the years different approaches and tools have been developed with the aim of helping the user during some of the optimization phases (monitoring, analysis or tuning phases). In Section 1.3 we present an overview of them.

The essential and, at the same time, most complex task during the tuning process is the performance analysis due to in practice, the bottlenecks can be found at different abstraction levels. Some of them can be caused by the use of *heterogeneous systems* rather than homogeneous ones or by dependences on the set of *input data*. These facts entail some potential bottlenecks which can vary through the execution or in different executions. Other problems, can come from communications, caused by an *erroneous conception of the application* -which can provoke some unexpected blocking in some communication functions-, the *communication library implementation* -the design or implementation of the software layers can be generic and not optimized to a particular system or conditions-, the *operating system characteristics* -the inappropriate size of a buffer or the management of the messages at protocol level can interfere in the message sending times- and the *underlying hardware capacities* -some characteristics of the interconnection network, such as latency or bandwidth, or even the volume of traffic in the network, can affect

the speed of the application execution-.

These examples show the complexity of the tuning process and the necessity for using automatic tools in order to simplify and accelerate the performance tuning process. However, even though in the performance optimization area there are several approaches and tools, in general all of them require the user to know parallel programming in-depth and take an active part in tuning the application. In consequence, tools capable of automating the tuning of parallel programs in a user-friendliness way need to be provided.

One of the available tools is MATE (*Monitoring, Analysis and Tuning Environment*), which is an automatic and dynamic tuning environment for parallel applications. As its name indicates, MATE works in three continuous and iterative phases in order to adapt the deployment of the application according to the current state of the execution environment. MATE includes the knowledge to tune performance problems in pieces of software called "tunlets". Each tunlet includes the logic to collect behavioural information, analyze it on the fly and decide what the required tuning actions are.

The objective of this work is to extend the usability of MATE. Our work covers two different aspects of MATE:

- the improvement of the performance reached by the centralized analysis executed by MATE, due to it turns in a bottleneck as the size of the application increases. Thus, we provide an alternative to provide MATE with scalability properties.

- the increase in the user-friendliness of MATE in order to facilitate the inclusion of new performance knowledge in it. Thus, we make MATE transparent for the users.

According to the first aspect, we propose a novel approach to execute the analysis process, called *Distributed-Hierarchical Collecting-Preprocessing Approach*. This approach is based on the distributed collection of events which alleviates the centralized old-fashion in which collection was done, and in the preprocessing of cumulative or comparative operations as possible. Thus, the *Global Analyzer* receives just the necessary information condensed

in a unique message from each *Collector-Preprocessor*, which considerably reduces the overload of Global Analyzer. In this way, MATE is provided with scalability properties. In order to validate this new approach we compared its deployment with the deployment of full centralized approach.

According to the second aspect, we provide a methodology, including a designed language and a developed translator to automatically insert tunlets (tuning techniques) in MATE. When some problem has to be tuned in a parallel application the user has to develop the corresponding tunlet. By using our methodology, the user is exempted from being involved in implementation details of MATE. Thus, by defining a set of abstractions about the application and the performance model, such abstractions can be formalized in a tunlet specification using the provided language. Such specification will be automatically translated in a tunlet ready to be used in MATE.

Chapter 1

Introduction

> "Una esperanza creía en los tipos fisonómicos, tales como los ñatos, los de cara de pescado, los de gran toma de aire, los cetrinos y los cejudos, los de cara intelectual, los de estilo peluquero, etc. Dispuesto a clasificar definitivamente estos grupos, empezó por hacer grandes listas de conocidos y los dividió en los grupos citados más arriba. Tomó entonces el primer grupo, formado por ocho ñatos, y vio con sorpresa que en realidad estos muchachos se subdividían en tres grupos, a saber: los ñatos bigotudos, los ñatos tipo boxeador y los ñatos estilo ordenanza de ministerio, compuestos respectivamente por 3, 3 y 2 ñatos. Apenas los separó en sus nuevos grupos (en el Paulista de San Martín, donde los había reunido con gran trabajo y no poco mazagrán bien frappé) se dio cuenta de que el primer subgrupo no era parejo, porque dos de los ñatos bigotudos pertenecían al tipo carpincho, mientras el restante era con toda seguridad un ñato de corte japonés."
>
> *Su fe en las ciencias*, Julio Cortázar

PARALLEL computing and performance tuning are two fields in constant evolution, given the increasing of the requirements for high performance computing. Through the years, several approaches and tools had been proposed and used in order to improve the behaviour of the parallel applications and to cover the expectations of the users. In this Chapter we present the general background of this work. We provide an overview of the main aspects of parallel computing and performance. Then, we present the different approaches to tune the performance of applications and summarize the related work. Finally, the aim of this work, the thesis contribution and the work organization are presented.

1.1 Parallel Programming aspects

Parallel applications development involves a set of additional aspects to the algorithmic ones. Such issues have to be considered in order to obtain a better behaviour. Just by considering the underlying parallel computer, there are several configurations among we can consider a system composed of a network of nodes, where there are some inherent delays in the data transmission. In addition, when the parallel computer is a heterogeneous or time-sharing system, the individual performance of each node can vary from one to other. As we will explain below, a problem can be parallelly solved when it has some characteristics; on the one hand, the problem could manage a considerably amount of data, then the data is divided to be parallelly processed by several processors. On the other hand, the problem could process the data by means of an algorithm comprising a set of relatively independent subtasks which could be assumed by different machines in order to operate like a serial factory. It is fundamental taking into account these characteristics and the underlying platform when designing the parallel program, in order to obtain the desired benefits. Thus, the *division* of the work in smaller parts and their corresponding *assignment* to the parallel nodes, are key aspects in designing parallel algorithms [24]. In the following, we introduce the main concepts involved in parallel processing.

Decomposition is the process of dividing the work in smaller parts, where all or some of them could be executed in parallel. Each portion of work is defined by the programmer and constitutes a **task** or computing unit. The number of tasks and their size determine the **granularity**. This depends rather on the **concurrency degree**, i.e., the maximum number of tasks which can be simultaneously executed. In general, the concurrency degree increases as the granularity gets finer.

Another issue to consider is the **load balancing** among the nodes, due to it is desirable that all of them have the same volume of work. The load balancing in a homogeneous environment can be obtained by fairly dividing calculus and communication. However, in heterogeneous or time-sharing environments, load balancing is a very difficult task.

One more aspect to take into account is the **scalability**. In general,

an application is scalable if bigger parallel systems can solve proportionally bigger problems in equivalent time, or smaller problems in a shorter period of time. Both concepts, *load balancing* and *scalability* are key aspects in order to achieve high performance computing in parallel machines, and are presented in more detail in Section 1.2.

Clearly, the development of parallel algorithms is a critical issue in solving problems. In practice, the design of a parallel algorithm could comprise some or all the following steps:

1. *Identifying parallelism* [18, 24].
2. *Choosing the decomposition strategy* [24].
3. *Choosing the parallel algorithm which will constitute the application* [24].
4. *Choosing the programming model and implementation interface in order to write the program* [18].
5. *Choosing the implementing style* [18].

In general, there exist several options for each step. However, in order to obtain an acceptable performance from the used resources only some of these options, the well-known ones, are usually considered.

The first step is needed to determine the portions of work which can be executed in parallel. The Bernstein's conditions [53] can be considered in order to define the meaning of "execute in parallel":

- **Bernstein's Conditions**: Given C_1 and C_2 two tasks, C_1 and C_2 can be executed in parallel without any synchronization if and only if none of the following conditions are true:
  1. C_1 write data which after are read by C_2 (read-after-write).
  2. C_1 reads data which after are written by C_2 (write-after-read).
  3. C_1 write data which after are re-written by C_2 (write-after-write).

The remaining four steps are closely related to each other, due to the decisions made in a certain step could affect the decisions in the following ones. In general, there are two different decomposition strategies to define how to divide the work in several concurrent pieces:

- *Data parallelism*: the data domain is divided into multiple regions which can be assigned to different nodes. Data parallelism is commonly used in scientific problems, due to it concurrently uses several nodes and exhibits natural scalability conditions.

- *Tasks parallelism*: the main parts of the program can be identified as tasks. Their parallel execution should be scheduled by considering the interdependences. Tasks parallelism is normally limited to low degrees of parallelism [29].

For both strategies there are different models of parallel algorithms, such as *Master/Worker* [33] for data parallelism and *Pipeline* [33] for task parallelism. In the case of programming models, the two main programming models were conceived in order to be used in the corresponding parallel architecture:

- *Shared memory*: the application data are in the global memory, which can be accessed from every node. This means that each processor can independently manage data everywhere in the memory. Some synchronization mechanism is needed to preserve the consistency and integrity of the shared data.

- *Message passing*: the data are associated to a particular node. To access to remote data nodes have to establish communications. In general, a process sends the data and another one receives it. Thus, sending and receiving primitives synchronize the program. PVM [21] and MPI [25, 26] are the most used message passing libraries.

The use of these two programming models is not restricted to the architectures they are inspired in. However, an alternative use of them can entail some performance degradation. The chosen programming model determines the selection of the programming language to implement the application.

1.2 Performance of parallel applications

Once a parallel application has been designed, implemented and tested, the quality of its execution has to be evaluated. The aim of this process is to evaluate the used mechanism. Frequently, parallel applications present execution values which do not cover the expectations. In such cases, the parts of the program responsible for the undesirable behaviour have to be isolated to find the causes. Unfortunately, the users are who should affront this problem. This requires the users to know which would be the desirable execution values. Thus, if the obtained values are under the acceptable limits, they are responsible for determining the parts of the program which should be measured and analyzed.

There are several causes for performance degradation, such as incongruences among application, software and hardware. Differences among communication bandwidth and processing speed or memory bandwidth can be some examples. Through the years, several indices have been defined in order to evaluate the deployment of parallel computing. Due to the complexity level involved, none of the simple measures is capable of provide a completely faithful measure of the system performance. This entails the use of diverse indices in order to measure different aspects.

In general, every performance study about a parallel program has to consider as parameters the execution time, the scalability, the efficiency, load balancing, memory requirements, throughput, network latency, input/output indices, network throughput, design cost, implementing cost, debugging cost, reusability, hardware requirements, portability and maintenance costs. The importance of each factor is relative to the nature of the problem.

Some of the performance parameters, such as the *execution time*, the *scalability*, the *efficiency* and the *load balance*, have a general importance which entailed the development of several mathematical models to formalize some of their qualities. These models allow the users to understand the general behaviour of the applications, and can be used to compare different executions or different implementations of the program.

Speedup. The main point of interest in developing parallel solutions is

to consider how fast is solved the problem [53]. The parallel execution time can be pondered by the execution time of a simple processor. If we define $T(x)$ as the execution time of an application with x processors, the speedup is a relative performance measure, defined as follows [18, 53]:

$Speedup(n) = \frac{T(1)}{T(n)}$

Thus, $Speedup(n)$ represents the speed increasing when a parallel system is used. However, the complexity of the program in parallel computing is not really representative, due to communications increment considerably the total execution time. The maximum possible speedup with n processors is n. Such speedup is reached when there is not additional overhead in using parallel computing and the computing can be divided in processes which last more or less the same and are assigned to different processors. In such cases, the speedup is linear.

$Speedup(n) \leq \frac{T(1)}{\frac{T(1)}{n}} = n$

Sometimes, the $Speedup(n)$ can reach values bigger than n, which is called *superlinear speedup*. This fact can be explained because the extra memory in the parallel system [53].

Scalability. The speedup of parallel applications can be limited by different factors. On the one hand, some regions in the program cannot be divided in concurrent parts. They have to be sequentially executed, such as the initialization and the finalization of the application. On the other hand, the application can present some idle periods in some processors, redundant calculations in each node or excessive interprocesses communication.

If the required time to process the sequential regions is denoted by f, the execution time with n nodes in parallel will be $f * T(1) + (1 - f)\frac{T(1)}{n}$. More intuitively, T_S denotes the sequential region and T_P denotes the parallel region. Figure 1.1 (based on [53]) shows a simple example where T_S comprises the initialization time and T_P comprises the parallel execution of the remaining part of the program.

The scalability is another of the interesting property, which is used to measure how efficiently a program uses the processors in a parallel system. This parameter provides an estimation of the profit taken from the involved resources. Particularly interesting is the evolution of the scalability when

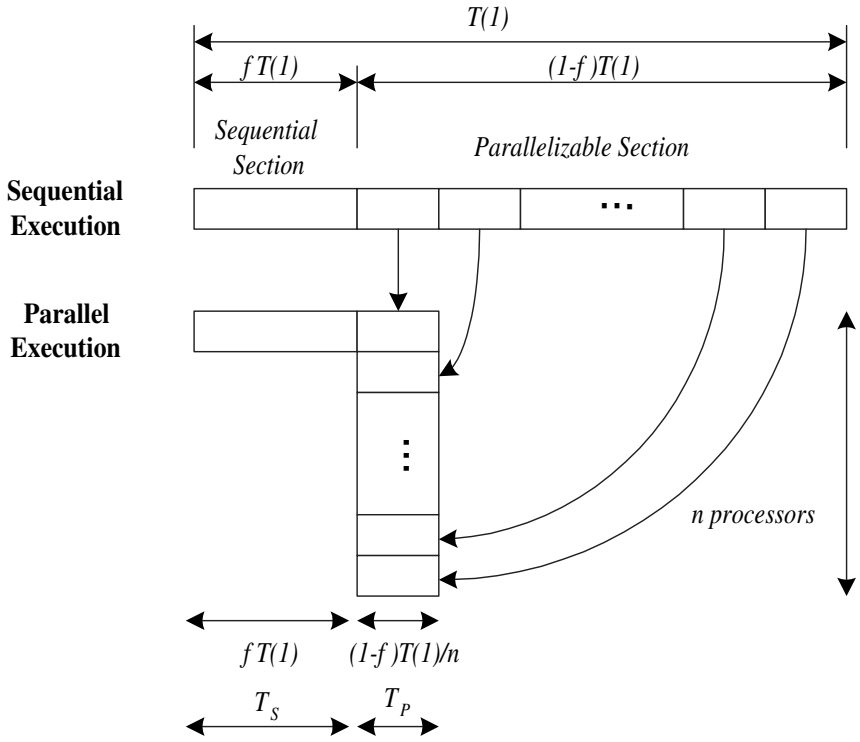

Figure 1.1: Parallelizing a sequential problem.

new processors are added to the execution. A series of laws in order to predict the scalability of applications have been defined:

- **Amdhal's Law**: Amdhal considered the sequential time T_S and the problem size as constant values. Then, even though an improvement in the speed can be observed, the concurrent computing fraction has to be a substantial part of the global computing in order to reach an increasing in the speed. This can be expressed as follows:

  $Speedup(n) = \frac{T_S+T_P}{T_S+\frac{T_P}{n}} = \frac{T(1)}{T_S+\frac{T_P}{n}} \leq \frac{T(1)}{T_S}$

  Thus, $\frac{1}{f}$ limits the maximum speedup, even though an infinite number of processors is considered. Amdahl used this argument along the 60th's in order to promote the uniprocessor systems. However, this law can be interpreted in a more positive way, by considering that the allowed speedup can be a big improvement.

- **Gustafson's Law**. Gustafson presented a new line in scalability con-

cepts, in order to demonstrate the Amdahl's law was not as much significant as supposed. In practice, a bigger number of processors allows for solving a bigger problem in a reasonable time. Then, the size of the problem can be determined by the size of the available set of processors. In this way, the execution time is considered as fixed in place of the problem size. In order to maintain constant the parallel execution time, the size of the problem has to increase as the size of the system increases.

By considering the parallel execution time T_P as a constant, the speedup is different from the speedup defined by Amdahl. This new magnitude is called **scaled speedup**, i.e. *speedup when the problem is scaled.* The parallel execution time is defined as $T_P = f*T(1)+(f-1)*T(1)/n = 1$. By applying an algebraic artifice, $T(1) = f*T(1) + (1-f)*T(1)$ is transformed in $n + (1-n)f*T(1)$ [53]. Then, the scaled speedup is defined as follows:

$$Speedup_S(n) = \frac{f*T(1) + (1-f)*T(1)}{f*T(1) + (1-f)\frac{T(1)}{n}} =$$

$$= \frac{n + (1-n)f*T(1)}{1} = n + (1-n)*f*T(1)$$

In this equation, there are two assumptions: parallel execution time and sequential execution time are constants.

In addition to the problem size constant scalability (Amdahl) and the execution time scalability (Gustafson), the scalability could present some memory problems, i.e. the problem is scaled as memory is available. In general, the size of the memory increases as the number of nodes increase. This can allow for the increasing in the execution time.

Efficiency. Sometimes, it is useful to know the use of the processors. This can be determined by using the *efficiency*, which is defined as follows [53] :

$Efficiency = \frac{Execution\ time\ using\ a\ simple\ processor}{Execution\ time\ using\ n\ nodes\ in\ parallel\ \times\ n}$

$Efficiency = \frac{T(1)}{T(n)\times n}$

In other way:

$Efficiency = \frac{Speedup(n)}{n} \times 100\%$

where the efficiency represents a percentage. For example, if the efficiency is of 50%, the processors were used, in average, during a half of the time. The efficiency reach 100% when $Speedup(n)$ is n.

Load Balance and load balancing. To execute a parallel program, the tasks must be mapped to processing elements. How the mappings are done can have a significant impact on the overall performance of a parallel algorithm. It is crucial to avoid the situation in which a subset of the processing elements is doing most of the work while others are idle. *Load balance* refers to how well the work is distributed among the processing elements. In an efficient parallel program, the load is balanced so each processing element spends about the same amount of time on the computation. *Load balancing* is the process of allocating work to processing elements, such that each element involved in the parallel computation takes approximately the same amount of time. The load balancing can be either static or dynamic, but in both cases the work is distributed as evenly as possible. [33]

1.3 Performance Analysis and Tuning

The main goal of parallel and distributed applications is to take profit from high computational capabilities of parallel systems. However, obtaining high performance of an application running in such a system becomes a hard task [38]. As explained in the previous section, there are several indices which allow for a general evaluation of different aspects of the applications behaviour. Nevertheless, none of them provides the users with specific information or suggestions to overcome the problems. Then, the development of an application with a good performance, forces the users to face the tuning process. Therefore, to attend the performance analysis problem and help programmers in the application improvement, many tools have been presented [38]. The basic purpose of performance analysis tools is to help a programmer to understand the performance characteristics of an application. In particular, the tool should analyze and locate parts of an application that exhibit

poor performance and cause program bottlenecks. Such tools are useful for understanding the behaviour of normal sequential applications and can be enormously helpful when trying to analyze the performance characteristics of parallel applications. Such tools are categorized as *monitoring tools*, *analysis tools* or/and *tuning tools*.

Most performance monitoring tools consist of some or all of the following components:

- a technology of inserting instrumentation calls to the performance monitoring routines into the user's application
- a run-time performance library that consists of a set of monitoring routines that measure and record various aspects of a program performance
- a set of tools for processing and displaying the performance data.

A particular issue with performance monitoring tools is the intrusiveness of the tracing calls and their impact on the applications performance. It is very important to note that instrumentation affects the performance characteristics of the parallel application and thus provides a false view of its performance behaviour [8].

With regard to the performance analysis tools, their objective is to automate the evaluation of the monitored information. In general, they include some performance knowledge to find bottlenecks and provide solutions. The complexity of the analysis process directed by the philosophy of the knowledge determines how fast the solutions or modifications are available to be introduced into the application.

Finally, the performance tuning tools attempt to automate the process of inserting modifications into the application with the aim of overcome the detected bottlenecks. Some tools cover more than one of these categories, helping the user in more than a simple level.

Every tool shares the goal of helping users to tune the behaviour of their applications. However, the manner in which this help has been provided has been continuously evolving. Thus, through the years, several approaches in performance monitoring, analysis and tuning have been proposed in order to

assist the users in improving their applications. In the following sections we provide an overview of them.

1.3.1 Classical performance analysis

The classical performance analysis approach is based on the *post-mortem* analysis of the application behaviour carried out by the user. The figure 1.2 presents the general flow of this kind of analysis.

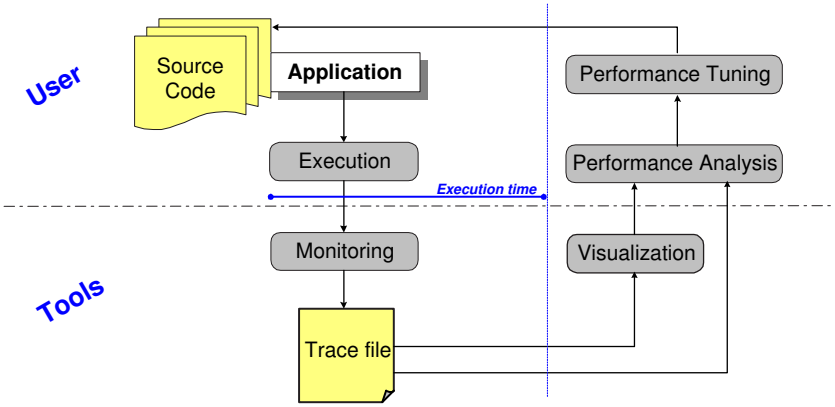

Figure 1.2: Classical performance analysis approach

First, while the application is running, a monitoring tool obtains information about the behaviour of the application. The corresponding instrumentation had been statically inserted by the monitoring tool or manually by the user. When the application is being executed and performing the instrumented code, the instrumentation allows for data measurements and collection. In the following step, this performance data are graphically interpreted by some visualization tool. Some of the common perspectives used to show the information are the Gantt, pie or/and bar charts and are used to present different views of message-passing, computing time, subroutines invocations, etc. During the performance analysis phase, the graphics are useful to help the user to understand the tracing in order to analyze the behaviour presented by the application through the execution. In the last step, the user manually changes the source code of the application in accordance with decisions made during the analysis. Then, the modified program has to be re-compiled and re-linked for future executions. This process is

successively repeated until an acceptable performance is achieved.

Even though the classical approach has been used for many years, it has several drawbacks. It requires the user to have a very high degree of expertise in order to analyze and make decisions on how to improve the behaviour of the application; this is a very difficult task due to the size of trace file is in general proportional to the size and the execution time of the application. In addition, visualization tools do not scale very well, which has as a consequence that when there is a high number of processes involved in the application or the execution time is too long, the graphics become unreadable. Furthermore, because of the analysis is made by considering a single execution, the tuning is only useful when the behaviour of the application neither depends on the input data nor varies from one iteration to another nor changes the platform in which it is executed. In summary, the classical approach constitutes a very time consuming task which is constrained to a reduced set of applications.

There are several tools following the classical approach. Some of them are only focussed on monitoring or on visualization, and some others combine both skills. In the category of *monitoring tools*, we can include Tape/PVM [32] and PICL [22, 67]; both of them generate trace files of PVM applications and require the recompilation of the application. **Tape/PVM** is a good base for some visualization or post-mortem analysis tools, and is focused on minimal overhead introduced into traced programs. **PICL** (*Portable Instrumented Communication Library*) is a subroutine library that can be used to develop PVM applications portable across several platforms. It provides a set of high-level communication routines and allows for enabling a mechanism of trace file generation. It is obsolete, but evolved to **MPICL** [64]: PICL for MPI.

In the category of *visualization tools*, we can mention ParaGraph [27, 66] and Vampir [45, 69]. **ParaGraph** visualized trace files in the PICL or MPICL format, but at present it only works with MPICL. It present the information by considering the processor utilization, the communication between processes and the task information. **Vampir**(*Visualization and Analysis of MPI pRograms*), monitors the application by using its own mechanism, VAMPIRtrace for MPI-based applications. It provides a variety of graphical

displays and filter operations to reduce the amount of managed information. It supports load balancing, analysis of performance of subroutines or code blocks, and identification of communication bottlenecks. It evolved to the **DAMIEN** (*Distributed Application and Middleware for Industrial Use of European Networks*) project [57] for Grid applications.

Some of the tools which integrate *monitoring and visualization* skills, are Pablo [48] and XPMV [21, 70]. **Pablo** includes application performance instrumentation, graphical (including 3D performance data representation) and sonic representation of the collected data and data amount reduction. **XPVM** is usually integrated with the PVM library and can be used as a graphical console, monitoring tool and post-mortem analysis tool.

1.3.2 Automatic performance analysis

The main contribution of the automatic performance analysis approach has been to release the user from having a high degree of expertise in parallel systems and performance analysis. This is shown in Figure 1.3.

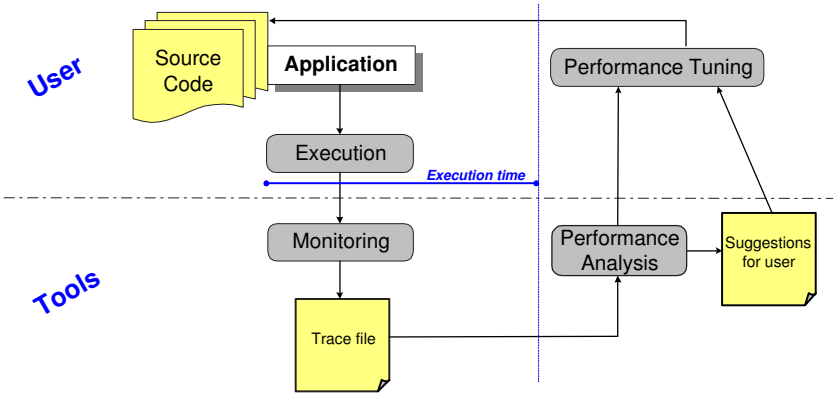

Figure 1.3: Automatic performance analysis approach

When the execution of the application finishes, the analysis tool looks for performance bottlenecks automatically by considering the information collected by the monitoring tool and its own knowledge about potential problems the application can present. Such knowledge is normally constituted by performance models of the well-known and typical parallel performance problems. The performance models allow for detection of bottlenecks as

well as their causes and needed changes to improve the future executions of the application. When the performance analysis finishes, it provides the user with the corresponding suggestions to modify the source code. As in the previous approach, the user changes the application, re-compiles and re-links it to next execution.

Although this approach exempts the user from the very difficult and time consuming task of analysing the behaviour of the application, it has some constraints. On the one hand it is still based on trace files and considers a single execution of the application; on the other hand, the creation of knowledge models is not an easy task and need a trade-off between simplicity and accuracy. Then, it is only suitable for the same set of applications as in the classical approach.

Some examples of tools following this approach are KappaPi [19], Paradise [31] and AIMS [54, 56]. **KappaPi** (*Knowledge-based Automatic Parallel Program Analyzer for Performance Improvement*), is based on Tape/PVM trace files. Its knowledge base includes performance models about the main bottlenecks found in message passing applications. The analysis is made by dividing the trace file in chunks which are separately analyzed. Regarding to **Paradise** (*PARallel programming ADvISEr*), it represents the execution of the program as an event-graph. It is viable for programs written in Charm++ [30], an extension of C++. Both, KappaPi and Paradise provide suggestions about the detected bottlenecks and the way to avoid them. A slightly different tool is **AIMS** (*Automated Instrumentation and Monitoring System*), due to it includes a source code instrumentor, a run-time performance-monitoring library, two tools that process the performance data (trace file animation and analysis toolkit)and a trace post-processor that removes overhead introduced by monitor. It can be used for FORTRAN or C message-passing programs written using the NX, PVM or MPI communication libraries.

1.3.3 Dynamic performance analysis

The dynamic performance analysis proposes to overcome the drawbacks presented by the automatic post-mortem performance approach, such as the

analysis step based on a single run of the application and on large trace files. The analysis is made "*on the fly*" by considering performance data collected by an *on-line monitoring tool*, which presents the benefit of independence from a trace file. Figure 1.4 shows the general view of this approach. The instrumentation can be dynamically inserted into or eliminated from the application by applying dynamic instrumentation techniques (explained in Chapter 2).

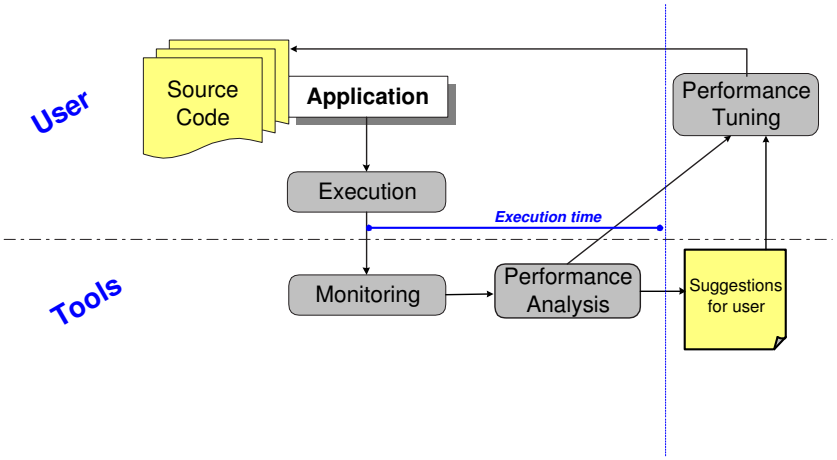

Figure 1.4: Dynamic performance analysis approach

The dynamic analysis approach allows for detection of performance problems faster than the *postmortem* approaches. It is suitable for iterative long-running applications. However, it requires the user to stop, modify, recompile and re-run the application in order to apply the tuning. Then, as in previous approaches, decisions based on a single execution could not be significant in future execution, when the application depends on the input data or their evolution.

Paradyn [37, 46, 65] is an example tool in the dynamic performance analysis approach. It is able to insert and modify instrumentation during run-time without any changes of the source code, due to it uses dynamic instrumentation (see Section 2.1.3). It has a special module so-called Performance Consultant which can be used in order to exempt the user form deciding which instrumentation is the most important, in order to minimize the intrusion inserted into the application.

1.3.4 Dynamic performance tuning

The dynamic performance tuning approach proposes to automate the insertion of modifications in the application. The previous three approaches have been incrementally overcoming the difficulties presented by their precedent approaches, dynamic performance tuning offers automatic tuning during run-time instead of manual insertion of changes in the source code. Figure 1.5 shows the general operation of this approach.

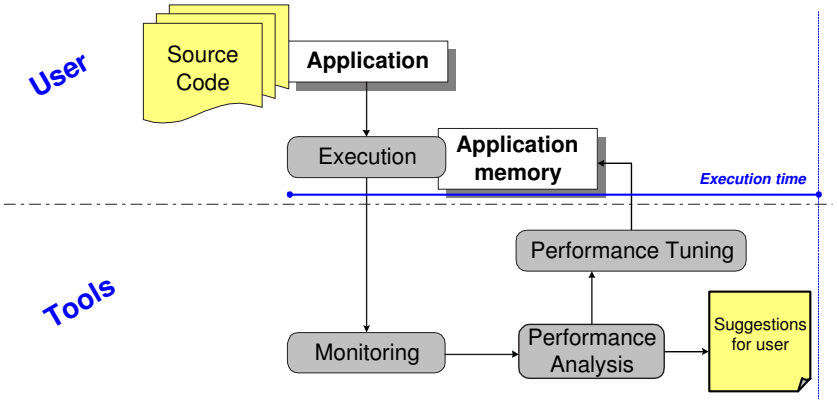

Figure 1.5: Dynamic performance tuning

All the phases in this approach are done during run-time. The analysis step is not based on trace files, but it uses the measurements provided by a dynamic monitoring tool. Depending on the evaluation of the performance, the tuning actions are automatically and dynamically inserted in the application. Thus, the running parallel application would be automatically monitored, analyzed and tuned on the fly without need to re-compile, re-link nor restart it. This completely exempts users from taking part in tuning their applications. Another advantage is that the performance of the application is evaluated and tuned according to its current behaviour in the environment. Then, decisions are more accurate and consistent, due to every execution of the application is separately tuned according to its particular execution conditions, i.e. different input data or different conditions in the execution environment. Such as in the previous approach, this one is suitable

for iterative, long running and resource-intensive programs.

There exist several tools in this approach, such as SCIRun [47, 68], Autopilot [49], Active Harmony [52, 55], AppLeS [6], Mojo [4], Dynamo [16], PerCo [35], Java HotSpot [63] and MATE [38, 39, 40]. Due to MATE is the core of this work, we dedicate an overview of it in the following subsection, and mention what makes it different from the others tools. In Chapter 2 we provide more details about MATE.

SCIRun (*Scientific Computing and Imaging*), is a problem solving environment (PSE) and a computational steering system in which large scale simulations can be processed. It allows to interactively steer a computation changing parameters, recomputing and then re-visualizing. First, a simulation can be composed via a visual programming interface to a dataflow network. Then, such a simulation can be executed, controlled and tuned by interacting with the end user via a graphical user interface. Finally, the information can be displayed using 3D graphics.

Autopilot, automatically chooses and configures resource management algorithms based on application request patterns and observed system performance. Its infrastructure is based on Pablo tool. It provides a set of *performance sensors*, *decision procedures* and *policy actuators*. The toolkit uses distributed sensors to gather quantitative and qualitative performance data from executing applications. Autopilot relies on fuzzy sets and uses a set of IF-THEN production rules that map the sensor input values to the actuator output space.

Active Harmony, is a framework which enables the dynamic adaptation of an application to the network and resource capabilities. It permits automatic adaptation of algorithms, data distribution, and load balancing. The system provides Library Specification Layer with uniform API. This layer integrates different libraries with the same or similar functionality. The user develops an application using this API, and hence the application contains a set of libraries with different algorithms and tunable parameters to be changed. During runtime Active Harmony monitors underlying library execution and manages the values of the different parameters. The system is able to select more efficient library and change tunable parameters to improve the application performance.

AppLeS (*Application Level Scheduler*), combines dynamic system performance information with application-specific models and user specified parameters to provide better schedules. It is developed on agent-base methodology. Each application has its own AppLeS agent. Each one contains static and dynamic information about the available resources and its function is to determine an application-specific schedule and implement that schedule on the distributed resources on metacomputers.

Mojo and **Dynamo**, perform the run time optimization of a native instruction stream. The program binary is not instrumented and is left untouched during the system operation. Thus, they uses very low-level techniques of optimization.

PerCo (*Performance Control*), can be used for distributed applications executing on a heterogeneous network, such as a computational Grid. The PerCo system is oriented to two HPC application domains: coupled models for scientific simulation [3] and distributed search for statistical disclosure control [34]. PerCo is capable of monitoring the progress of the applications and redeploying them so as to optimize performance. PERCO requires performance prediction capabilities, such as history of previous executions.

Java HotSpot provides the highest possible performance for Java applications. Traditionally bytecodes are generated from Java programs and then interpreted during execution by Java Virtual Machine. Java HotSpot includes dynamic compilers that adaptively compile Java bytecodes into optimized machine instructions and efficiently manages the Java heap using garbage collectors, optimized for both low pause time and throughput. It provides data and information to profiling, monitoring and debugging tools and applications.

MATE

MATE (*Monitoring, Analysis and Tuning Environment*) [38] is a tool which implements automatic and dynamic tuning of parallel applications. The operation of MATE is based on three automatic and continuous phases along the application execution: monitoring, analysis and tuning. The knowledge about what to measure, how to evaluate the behaviour and what to change

to adjust the behaviour is based on performance models managed by the Analysis phase. Analysis phase indicates to the Monitoring one, what is the information needed to evaluate the model by considering its parameters. In consequence, such phase inserts the corresponding instrumentation. Collected information is sent to the Analysis phase as events. Then, the performance model can be evaluated. Depending on the decision, the Tuning phase will receive a requirement for changing something in the application. Each performance model is encapsulated as a piece of software called "tunlet".

Even though MATE shares some characteristics with the tools previously presented, it has some particularities. On the one hand, if we consider the preparing of the application to be tuned, using MATE the monitoring is based on the dynamic instrumentation where the application does not require to be prepared for tuning due to measure and tuning points are inserted on the fly. In Autopilot the developer must prepare the application inserting sensors and actuators manually into the source code. In Active Harmony the mechanism is based on the integration of different libraries with the same functionality.

On the other hand, if we consider the way in which the performance analysis is made, MATE uses simple, conventional rules and performance models, whilst Autopilot uses fuzzy logic to automate the decision-making process, Active Harmony uses heuristic algorithms in order to describe the application behaviour, and PerCo is based on history. In addition, MATE is focussed on the efficiency of resource utilization and performance bottlenecks that occur during the application execution, while AppLeS is focused on the resource scheduling. Finally, MATE works at binary program level rather than at native instruction stream level such as in Mojo or Dynamo.

A more detailed description about MATE is presented in Chapter 2.

1.4 Thesis Contribution

Our proposal is to enhance and extend the use of MATE from two different sides. The first point is related to make MATE scalable, with the proposal of overcome the bottleneck presented by MATE when the number of machines involved in the execution of the application increases. The second point is

relating to automate the creation of tunlets (the inclusion of knowledge in MATE) in order to make easier the use of this environment.

To start with the first point, we propose a new approach to perform the analysis phase of MATE. The novel approach is called the *distributed-hierarchical collecting-preprocessing approach*. Until now MATE had been following a centralized approach, in which the collection and processing of the information turned in a bottleneck as the amount of processes in the application -and consequently the amount of events- increased. Thus, such approach limited the scalability properties of MATE. Then, we studied different options to provide scalability; however, both distributed and hierarchical approaches presented constraints from the user, the performance model and the application point of view. Thus, we selected the good characteristics of them in order to provide a viable alternative. The objective of the proposed approach is to overcome the bottleneck of Analyzer, by distributing what can be distributed (the collection of events) and preprocessing what can be processed before the model evaluation. We compare the new approach with the centralized one in order to appreciate the significance of the contribution. We also study the intrusion caused by MATE and the resources requirements.

Concerning to the second aspect of the extensions of MATE, we define a methodology to specify performance models according to applications. The purpose of such methodology is to palliate the constraints imposed until now to use MATE, due to the users had to program their tunlets by hand considering the implementation details of MATE. Then, this part of the present work represents an important contribution from the usability and user-friendliness point of view, owing to from now the user will be only concentrated in the application and the performance model. By writing the specification of the performance problem, the user can automatically access to the possibility of using such knowledge in order to tune the application: the specification is automatically translated in a tunlet to be used in MATE to dynamically and automatically tune the application. In order to define the methodology we studied the viability of defining some enough expressive language in order to formalize the specifications. As a result, we designed and defined a context free language, and developed a translator to create

tunlets from specifications.

We also provide two use cases of such methodology, which at the same time provide MATE with two new tunlets to tune Master/Worker applications. The developed tunlets allow for tuning the number of workers and the load balancing, respectively.

1.4.1 Work organization

This work is organized as follows: in Chapter 2 we give a general description about MATE and summarize the main aspects of dynamic instrumentation. Then, we explain in more details the main aspects relating to the architecture of MATE.

In Chapter 3, we describe the proposal to provide Analyzer -and in consequence MATE- with scalability. We include the comparison with centralized approach and the study of the intrusion of MATE.

In Chapter 4, we describe the methodology to automatically generate tunlets from specifications. Then, we present the process followed in the definition of the designed Tunlet Specification Language and the developed Translator. At the end of the chapter, we summarize the main aspects to be taken in consideration by the users. The complete syntax directed definition of the language is presented in Appendix A.

Chapter 5 is dedicated to depict some examples and use cases of the methodology presented in Chapter 4. The examples include the complete specification of the tunlets, documented in Appendixes B and C.

Finally, the main conclusions and open lines that can extend this research are reported in Chapter 6.

Chapter 2

Monitoring, Analysis and Tuning Environment

> "En El Cairo uno entra en una tienda y le ofrecen, inmediatamente, café, vino, frutas... Luego le dicen '*Bienvenido a Egipto*' . Después cuando uno pregunta el precio de algo, con toda cortesía le advierten. '*¡No Señor! ¡Es un regalo!*' Pero se sobreentiende que esto es una convención y que no es un regalo que se daba aceptar. En seguida viene el regateo, que puede durar media hora o tres cuartos de hora. Uno ofrece cinco y ellos piden veinticinco y todo eso para que, finalmente, el precio quede en diez. Y es una maravilla porque si uno no compra nada, igual son muy corteses. Ellos no han descubierto el mate, pero igual han encontrado una manera, casi más simpática, de perder el tiempo"
>
> *Borges, sus días y su tiempo*, Jorge Luis Borges

MATE , which means Monitoring, Analysis and Tuning Environment, provides dynamic and automatic tuning of parallel applications. The power of this tool lies in its two characteristics:

- *Dynamic tuning*, is useful especially when applications are executed in **heterogeneous** or **time-sharing** systems, because decisions to adjust the behaviour of a determined application are made on the fly, taking into account the current state of the system.
- *Automatic tuning*, is helpful because users have not to be worried nor envolved in looking for performance bottlenecks nor applying solutions

into the applications to improve their performance. They only have to statically cooperate with the tool to indicate what the performance problems the application can present are.

Decisions on how to improve applications are made by considering performance models of possible problems the applications could present. Then, some instrumentation to collect information about the behaviour of the application is automatically inserted according to the model. Analysis of the gathered information is made evaluating the given performance model. Solutions are automatically inserted in the application, and applications do not need to be re-compiled, re-linked or restarted.

Dynamic tuning implemented by MATE is the core of this work. Therefore, in the following sections, we describe in more details the main aspects of MATE (more details can be consulted in [38, 39, 40, 41]). We provide its general characteristics, functionality and different techniques used during the MATE development. The original version of MATE presents some limitations and restrictions to be used. On the one hand it has some difficulties when the number of machines increases. On the other hand not only the cooperation of the user is needed to define how to analyze the behaviour of the application, but the user must also know the implementation details of MATE to make possible the implementation of the solutions. The proposed enhancements and improvements are presented in the next chapters.

2.1 General view of MATE

In this section we present MATE in a general and conceptual way, in order to understand what its philosophy is. We describe the tuning approach applied in MATE, and the mechanism it uses in order to implement dynamic instrumentation and tuning.

2.1.1 Dynamic tuning

In Chapter 1, we presented the different tuning approaches. As commented, MATE implements dynamic and automatic tuning. It is composed of different services cooperating among themselves:

- *Dynamic monitoring of the application execution.* This service provides the metrics obtained during the execution. The instrumentation of the application, i.e. the insertion of code to examine the application behaviour of the application, is performed automatically to reduce and make easier the intervention of the user. Collected information is directly sent to the analysis phase.

- *Automatic performance analysis "on the fly".* This service analyzes the received information in order to detect possible bottlenecks and to provide solutions to overcome them. Performance knowledge about significant parallel bottlenecks is needed to detect problems and provide solutions.

- *Automatic tuning of the application during run-time.* This service is responsible for automatic insertion of the solutions provided by analysis phase into the application during its execution. This kind of tuning exempts the modification of the code due to the application is modified on the fly.

These services make easier the users' task if we consider the intervention in the tuning process. On the one hand, the user does not need to instrument the application by hand or semi-automatically, nor to trace the execution of the application, nor to analyze analytically or automatically the performance nor to modify and re-compile the application source code. On the other hand, this tuning approach is not restricted to homogeneous applications with a regular behaviour, it is suitable for applications that are executed under different conditions.

2.1.2 Additional characteristics

Additionally to the characteristics straightforward adopted from dynamic tuning approach, MATE considers the following aspects in order to work as a whole:

- *Parallel control of the application.* Control and Optimization services, i.e. monitoring and tuning respectively, should act over every machine

involved in the application execution in order to manage the entire application.

- *Global Analysis.* Even though the application is executed as subtasks in separate machines, the improvement of local tasks does not necessarily means an improvement in the whole application. Then, the behaviour of the application has to be evaluated in a global manner. Hence, some information about the individual tasks is collected and centralized to perform global analysis.

- *Application knowledge.* Dynamic tuning is executed during run time. This has two requirements:

  - simplicity in the analysis process, to make decisions in a short period of time,
  - conciseness in modifications to be inserted in the application.

  These two factors restrict the usability of dynamic tuning. Furthermore, the effectiveness of dynamic tuning could be decreased when no knowledge about the application is available. This presents the need of providing information about what should be measured, how to detect and solve existing bottlenecks and what to modify. Another difficulty is the representation of the knowledge.

- *Monitoring and optimization during run time.* As mentioned before, every phase in monitoring, analysis and tuning process has to be done "on the fly". The key is to determine the insertion of instrumentation and changes in the program without access the source code of the application. To make it possible, **dynamic instrumentation** is used. This technique allows for inserting a piece of code in a program in execution. One of the advantages of this approach is that the instrumentation can be inserted or removed as necessary. In fact, MATE uses **DynInst** [7], a library which implements dynamic instrumentation, both to instrument and modify the application. We provide an overview of this library in Section 2.1.3.

- *Low intrusion.* The overhead caused by the continuous monitoring, analysis and tuning processes should be minimal to avoid affect the performance of the application, due to they are executed concurrently with the application. Monitoring should manage reduced amount of information, analysis process should not be too complex, and tuning phase should not comprise very significant changes in the application.

- *Overcome the bottlenecks.* The monitoring, analysis and tuning processes require a certain period of time to provide an applicable solution to the existing problem. This can present a disadvantage: when the solution is ready to be applied, the application bottleneck may disappear. This fact could be a restriction because problems which appear isolately could not be effectively solved. This is why dynamic tuning is applicable to problems which have certain persistence along the execution.

2.1.3 Dynamic Instrumentation: DynInst

> *"The normal cycle of developing a program is to edit source code, compile it, and then execute the resulting binary. However, sometimes this cycle can be too restrictive. We may wish to change the program while it is executing, and not have to re-compile, re-link or re-execute the program to change the binary."* [7]

The principle of the dynamic instrumentation consists in to postpone the instrumentation of the application until it is being executed. Instrumentation can be inserted, modified or removed as needed during run time. DynInst is an API (*Application Program Interface*) which generates code during run time. As explained below, this library is used by MATE in order to manage the instrumentation of the application. In the following, we summarize the main concepts inherent in DynInst. More details can be obtained from [7].

DynInst provides a C++ library to dynamically instrument independent machine code. The API is based on object oriented technology, and provides a set of classes and methods which allow the user to perform the following actions:

- To modify an executing process or to start a new one.
- To create a new piece of code.
- To access and use existing code and data structures.
- To insert created code into executing programs.
- To remove inserted code from executing programs.

The instrumentation inserted in the application will be executed when the program executes another time the modified block. The program has not to be re-compiled, re-linked nor restarted. Due to DynInst manages the address space image of the process, the library does not need to access to the source code of the program. A fundamental requirement of DynInst is the debug information, which is used to locate the functions and variables in the application. Because of this, the programs to be instrumented must be compiled using the correspondent compiling option.

Library abstractions

DynInst is based on the following abstractions:

- **Mutatee** or **Application**: this is the program to be instrumented.
- **Mutator**: this is the program which controls and modifies the mutatee via DynInst.
- **Point**: this is an specific point in the application where some new code could be inserted. Examples of points can be the entry or the exit of a function.
- **Snippet**: it is a representation of a piece of executable code, which can be inserted into a program at a determined point. A snippet can include conditionals, function calls, loops, etc.
- **Thread**: this corresponds to a thread of execution.
- **Image**: this constitutes the static representation of a program in the disk. Each thread is unequivocally associated to an image.

The abstractions used by DynInst and their interactions are shown in Figure 2.1. In particular, the snippets are used to implement the data collection. An example can be the programming of a function call counter.

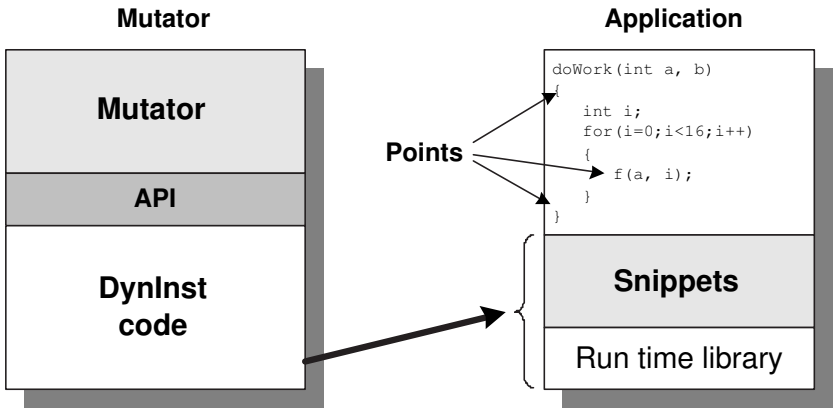

Figure 2.1: DynInst API abstractions

Using DynInst

To be able to dynamically insert instrumentation into an application, a user of DynInst library has to attend the following steps:

1. Have a mutatee executable file. Neither the source code of the application is needed nor special compiling or linking options besides the debugging option.
2. The mutator has to be implemented by using the appropriate classes of DynInst.
3. The mutator has to implement snippets by using the classes provided by DynInst.
4. The mutator has to be compiled and linked to DynInst library.
5. The mutator has to be executed.

Subsequently, DynInst executes the following steps during run time:

1. DynInst library is loaded in the address space of the mutator.

2. Via DynInst, the mutator creates a process to execute the application.

3. DynInst automatically loads its library and the snippets in the address space of the mutatee process.

4. DynInst inserts snippets calls in the specified points of the application.

5. When a function has a snippet call, for instance in its entry point, the code of the snippet is firstly executed and then the original code of the function.

The MATE environment uses DynInst to be able to instrument the application. MATE acts as the mutator whereas the user application acts as the mutatee. Snippets and Points depend on the information necessary to evaluate the behaviour of the application. Snippets are the pieces of software used to obtain the information and make the events. The points are the locations in the application that allow for collection of the information about the application behaviour. In the following sections, we explain the architecture an operation of MATE in more details.

2.2 MATE

MATE (Monitoring, Analysis and Tuning Environment) is, as its name indicates, an environment which provides dynamic and automatic tuning of parallel/distributed applications. The steering of the application comprises three different phases: monitoring of the behaviour of the application, performance analysis and tuning. All these phases are continuously and automatically executed on the fly. The main goal of this tool is to improve the performance of an application, by adapting it to the variable current conditions of the system. Hence, the user is exempted from manual application tuning.

In order to dynamically and effectively optimize the applications, the considerations made in sections 2.1.1 and 2.1.2 for dynamic and automatic tuning are implemented by MATE. As shows Figure 2.2, MATE instruments the application during run time to obtain information about its behaviour. The analysis phase receives the collected data as events, looks for possible

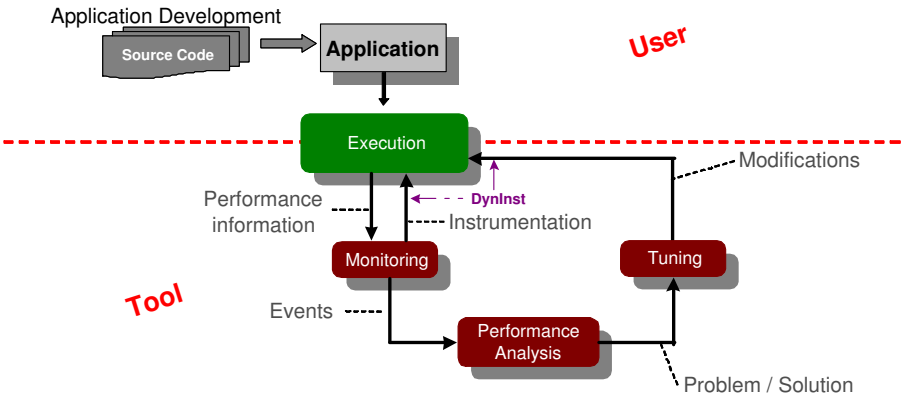

Figure 2.2: Operation of MATE

bottlenecks and tries to find solutions to overcome the problems. The solutions are inserted into the application by the tuning phase. The run-time changes of the application, for both the monitoring and tuning processes, are implemented via DynInst.

2.2.1 Architecture

MATE is composed by several components which cooperate among them to control and to improve the execution of the application. The main components are the following:

- **Application Controller** (*AC*): it is a daemon like process which controls the execution and dynamic instrumentation of the individual tasks.
- **Dynamic Monitoring Library** (*DMLib*): this is a shared library which is dynamically loaded in the application tasks. It is used to perform the data monitoring and collection.
- **Analyzer**: this process carries out the performance analysis of the application. In addition, it decides what have to be monitored and tuned.

Figure 2.3 represents the basic structure of MATE in a PVM scenario. For conciseness reasons, only the main components have been included. In

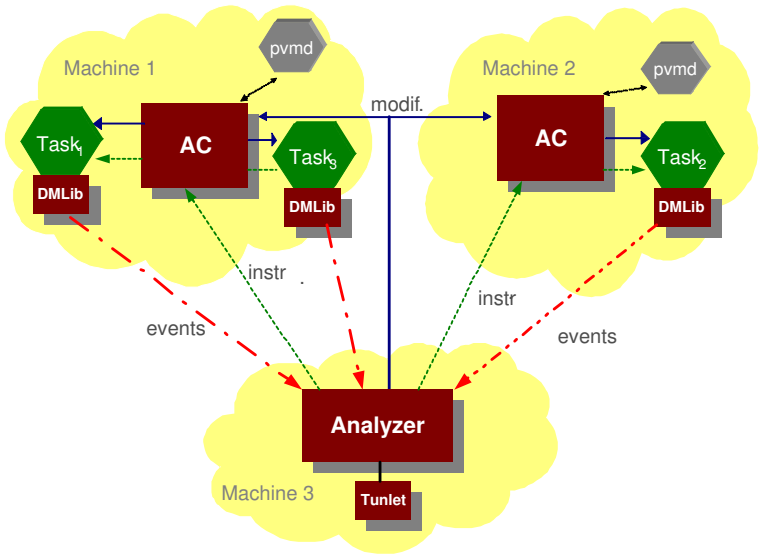

Figure 2.3: Architecture of MATE

this example, the application is divided into three tasks distributed among two different machines. When the execution starts, MATE distributes AC processes to both machines to control the start-up of the tasks. The AC master resides in the same machine as the PVM master daemon. It provides the control of the virtual machine, the creation of local tasks and the synchronization of the clocks of the different hosts. The slave AC is executed on the other machine, to control the creation of local tasks and synchronize the local clock with master AC's. In order to control the creation of tasks, both ACs communicate with the local PVM daemon (*pvmd*). When a new PVM task is started, DMLib is loaded into the memory of the task to provide its instrumentation. During run time, ACs manage the instrumentation of each task. This allows Analyzer to dynamically add or remove events. DMLibs are responsible for sending events to Analyzer. When an iteration has finished, Analyzer looks for bottlenecks by analyzing the obtained information. If some change in the application is required, Analyzer asks the AC for tuning. This is the general description of the cooperation among the different parts of MATE. In the following sections, we provide some additional description about each particular module.

2.2.2 Application Controller - AC

The Application Controller or AC is a daemon like process which controls the execution of the application, manages the dynamic instrumentation of the application tasks and supervises the modifications of the application during tuning phase. AC is composed of the Monitor and Tuner modules that cooperate among them to provide the required functionality. More details can be consulted in [38].

Monitor

Monitor is the module responsible for monitoring the execution of an application. The monitoring is based on function calls event tracing. The application is dynamically instrumented and the inserted instrumentation generates events. When MATE is launched, Analyzer indicates to Monitor the set of events to be traced. Conceptually, events are called *measure points*. When the application execution starts, Monitor inserts the code needed to catch the events into the running application.

The instrumentation could vary during run time. To find bottlenecks Analyzer may need some additional information, or may need to remove some useless instrumentation. Monitor is notified by Analyzer if some changes in the instrumentation are needed. In consequence, Monitor supports the modification of the set of monitored events.

DynInst is the library used by Monitor in order to carry out dynamic trace of events. For example, the instrumenting code can be inserted in the entry or exit of the sending or receiving functions to monitor the network characteristics; this information can be useful to analyze if there exist some bottleneck inherent in the communications.

The events are collected and sent to the Analyzer process by using DMLib -explained in Section 2.2.3, which was loaded during the start-up of the task. The communication with Analyzer is established by using an event collection low level protocol based on TCP/IP.

Monitor has to dynamically create instrumentation code -or in terms of DynInst so called *snippet*- for each new event to be traced. When the event happens, the snippet obtains from the parameters of the function and

the global variables all the attributes associated to the event. A particular snippet is executed each time the function it is inserted in is executed. The generated event is passed by DMLib to the Analyzer.

Tuner

Tuner is the module responsible for applying the tuning actions over the application tasks. The needed changes are determined by the solutions proposed by the Analyzer. Tuner modifies the application execution via DynInst, by modifying the memory associated to the application. Tuner provides an API which defines the set of tuning actions that the Analyzer can require:

- *LoadLibrary*: this loads a certain library in a process. This allows Analyzer to load additional code required to the tuning process.
- *SetVariableValue*: the value of a certain variable in a determined process can be modified.
- *ReplaceFunction*: this allows to replace every call to a certain function for a call to another function.
- *InsertFunctionCall*: a new function call with its attributes can be inserted.
- *OneTimeFunctionCall*: it allows to call a certain function once during the execution.
- *RemoveFunctionCall*: every call to a certain function is eliminated.
- *FunctionParamChange*: the value of a parameter can be changed in the entry of a function, before the body of the function is being executed.

Every tuning action includes a synchronization parameter called *breakpoint*, which specifies *when* the tuning action should be executed to ensure the correctness of the application behaviour. The breakpoint is inserted in a specific point of the application. When the execution of the application reaches that breakpoint, the tuning action is executed and the breakpoint is removed.

2.2.3 Dynamic Monitoring Library - DMLib

DMLib is a dynamic library which provides event tracing. The AC process loads this library on the address space of every process in the application in order to simplify the instrumenting and data collection. The library includes a set of functions responsible for providing events with all associated attributes, according to the Analyzer requirements. The DMLib API includes functions to perform the following operations:

- *To initialize the library*, by providing information about the process to be monitored, the Analyzer host location and the differences in clocks. DMLib establishes a connection via TCP/IP with Analyzer that allow Analyzer for recognizing, receiving and processing the events incoming from DMLibs.

- *To finalize the library*, which should be the last action to be invoked. It releases all the acquired resources, notifies the Analyzer process that the application processes have finished and closes the connection with it.

- *To register events.* The identifier or name of the event has to be provided and also the specific function and point in which the event should be caught. If the event has some associated attributes, the description of them (this is the data type and the identifier) has to be provided. When the register of an event finishes, it means that this kind of event can be sent to the Analyzer. Thus, the Analyzer will be able of obtaining the needed information.

Monitor creates a snippet which is inserted in the application when the registering of a new event is required. In this way, when a snippet is invoked all the attributes associated to the event are obtained, and the event is registered via DMLib. Each event includes at least *timestamp, event identifier, number of attributes,* and *attributes.*

DMLib uses a set of buffers in order to minimize the network overhead when sending events. The sending of events is controlled by time to avoid excessive wait for individual events.

2.2.4 Analyzer

The Analyzer is the module which governs the tuning of the applications. It requires the necessary metrics, carries out the performance analysis, and requires for changes in the application. In order to be able to evaluate the deployment of a given application, Analyzer requires both some application knowledge and the online event tracing.

From a functional point of view, the Analyzer is divided into two main parts which are the ***Dynamic Tuning API (DTAPI)*** and the ***Tunlets*** [38].

Dynamic Tuning API

The *DTAPI* constitutes the interface to handle the improvement of the application performance. It is the part which encapsulates all the low-level issues related to controlling the execution of the parallel application, i.e. its performance monitoring and tuning. As will be explained in Section 2.2.4, tunlets use the DTAPI as an interface in order to require the instrumentation necessary to evaluate the performance model; in order to be able of receiving and processing such events, the tunlets have to be associated to each event as an *event handler*. In the next, we intend to explain these issues.

DTAPI is implemented as a distributed asynchronous system where:

- the monitoring instrumentation and tuning service requests are delegated to distributed Application Controllers that in turn instrument and tune the application tasks
- the incoming events (event records sent by DMLibs and meta data sent by ACs) are collected and dispatched to registered event handlers.

The Dynamic Tuning API is provided as a collection of C++ classes. Most important of them are illustrated in Figure 2.4.

From the user of MATE point of view, these classes can be conceived as follows: *Application* object represents the analyzed application which consists of a number of *Tasks*. Each *Task* represents an individual application process (i.e. PVM task) and contains meta data (properties specific to that

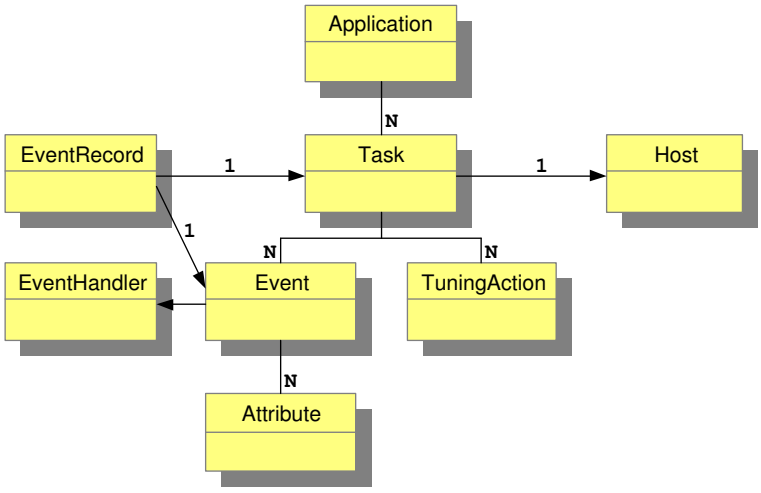

Figure 2.4: Dynamic Tuning API class diagram

task, e.g. process identifier, host where the task is running, etc. Each task may have a number of events to be monitored. A traced event is represented by the *Event* object. An event contains a set of *Attribute* objects that define what information should be recorded when the event is caught. Each *Event* object is associated with an *Event Handler* that is called each time a record of the event occurrence is received by the Analyzer. Therefore, as mentioned before, due to tunlets are associated to events as *event handlers*, each time a specific event is received, it is handled by every handler -tunlet- associated to it. In addition, the *Task* object contains the history of all tuning actions performed on that task.

In more depth, these classes present relevant information from the descriptive point of view of the application and the information needed in order to tune it: *Application* class includes general information about the application path, status and set of task, monitored events, etc., and provides a variety of methods to start the application or instrument it both for monitoring or tuning. *Task* class contains the name, id, status and monitored events among others, and has methods analogous to the *Application* class, but in this case to instrument the individual tasks. Every object of the *Event* class comprises data about the exact point in the code where the event must be caught (the function or the combination of class and method), and the

attributes associated to it. Each *Attribute* object es defined by its properties, such as data type and source. The *EventRecordClass* encapsulates complete information about the event, when and where it took place. For clarity reasons, we have presented the most significant classes. However there are many other classes that support the handling of the incoming events, managing the start and finish of a task, or controlling the hosts in the virtual machine.

Tunlets

The knowledge about a particular performance problem an application can present is represented by a tuning technique. A tunlet may be defined as a module of software which describes a particular performance problem of a running application, and provides the means to modify the execution to reach an optimal behaviour. As the Figure 2.5 shows, each tunlet defines and implements a particular tuning technique, i.e. the logic to overcome a particular performance problem by encapsulating the knowledge about the performance problem in the terms that follows:

- A set of **measure points** in order to indicate *what* is needed to detect the performance problem.
- A **performance model** to determine *how* to evaluate the collected information in order to detect bottlenecks.
- A set of **tuning actions** indicating *what, where* and *when* to change in the application execution in order to overcome the detected bottlenecks.

The tunlets should use Dynamic Tuning API to manage the application by invoking monitoring and tuning requirements, and to handle the events to collect the information of the application necessary for its analysis. Then, the DTAPI constitutes the interface that tunlets must follow to correctly work in MATE.

Owing to Analyzer manages the performance analysis process, it includes a set of tunlets which in fact provide the performance analysis logic. When Analyzer starts its execution, a particular tunlet indicates to the Analyzer what the set of measure points required to be able of evaluating the behaviour

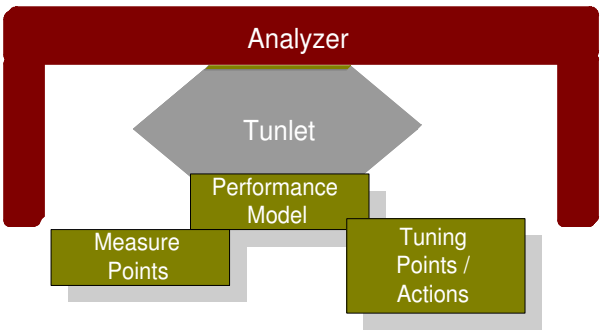

Figure 2.5: Elements of a tunlet

of the program is. Analyzer forwards the requirement to every AC. Then, Analyzer requires the master AC to start the execution of the application. When the application has started, the Analyzer enters in a bottleneck search phase. It continuously receives requested event records generated by different processes.

When an event record arrives to the Analyzer, the tunlet is notified in order to search for bottlenecks and determine their solutions. By examining the set of received event records, the tunlet extracts measurements and then it evaluates a built-in performance model to determine the actual and optimal performance. If the tunlet detects a performance bottleneck, it decides if the actual performance can be improved in existing conditions. If so, it then requests the Analyzer to apply the corresponding tuning actions. A request determines what should be changed (tuning point/action/synchronization) and it is sent to the appropriate instance of AC, and hence to the Tuner.

2.3 MATE as a development and tuning environment

In addition to its tuning properties, MATE is provided with a framework for the development of Master/Worker applications [10, 11, 43, 44]. The Master/Worker model of parallel algorithm consists of two logical elements: a *master* and one or more instances of a *worker* **??**. We represent the flow of this model in the Figure 2.6.

The master initiates the computation and sets up the problem. It then

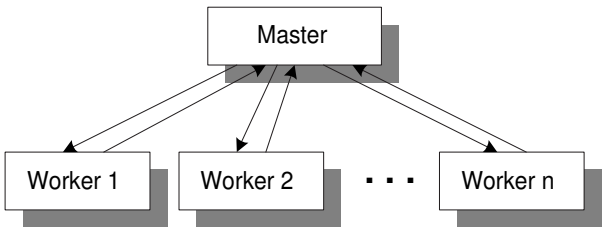

Figure 2.6: View of the Master/Worker parallel algorithm

creates a set of tasks to divide the work among the workers. The workers process the tasks and then send back to the master the results. In the classic algorithm, the master waits until the job is done, receives the results and finalizes the computing. This process could be iterative.

When users implement their applications by using this framework, they automatically can use MATE to tune the applications.

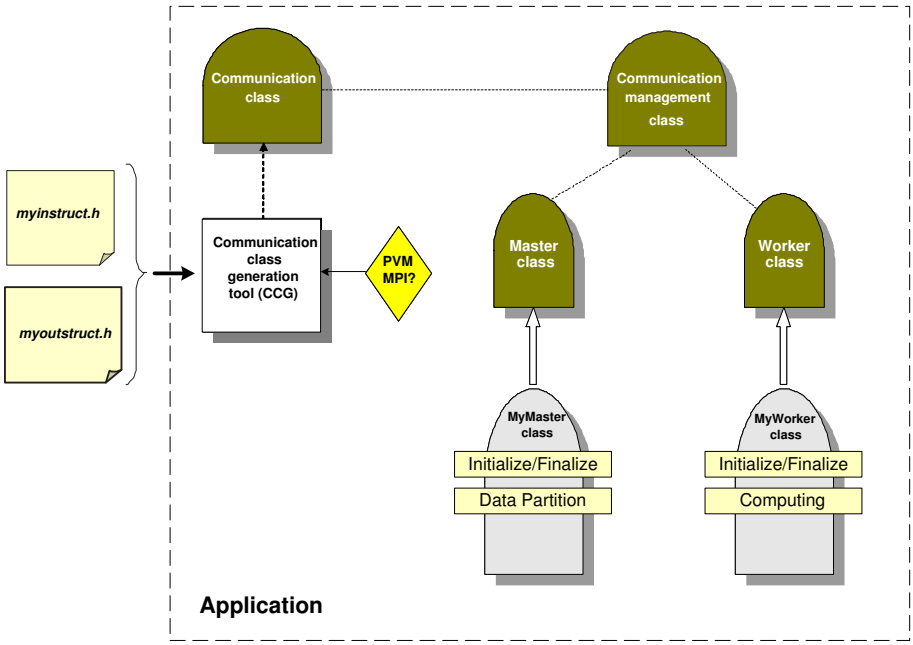

Figure 2.7: Class diagram of the Master/Worker Framework

The framework is constituted by general classes Master and Worker which encapsulate the communications and the general functioning of the Master/Worker programming model. The specific methods inherent in the pro-

blem solving are virtual methods, which the user has to define. As shown in Figure 2.7 the user has to provide the input and output data structures. The first one indicates the data structure of the input data to be processed along the application. The second one indicates the structure in which the results of the processing have to be stored.

In addition, the *Initialize* and *Finalize* methods are to be defined for master and worker. The master process requires another method -*Partition*- to make the partition of the data among the workers and the worker class needs the *Computing* method.

Due to MATE provides some tunlets defined over the framework, (this is the measure points, the performance functions and the tuning points are depending on the framework classes) every application developed using this framework can be automatically tuned by those tunlets. This represents a great benefit specially for non expert users: they not only can develop their Master/Worker applications without been worried about communications, but they also can automatically tune them according to the problems the tunlets can overcome.

At the moment MATE offers only the Master/Worker framework [36], but in the future, the idea is to incorporate new programming models, and skeletons, and tunlets to tune them.

Chapter 3

Scalability of Analyzer

> "Es entonces cuando aparece Eddington con una teoría revolucionaria. Guiado por la idea de que la palabra expansión se refiere a algo esencialmente relativo, atacó el enigma desde un punto de vista nuevo. Cuando decimos que el universo se expande, queremos significar que se agranda con relación a algo de tamaño constante, por ejemplo, con respecto al metro de París. Esta clase de expresiones tiene un valor relativo: Gulliver es un gigante al llegar a Lilliput y se convierte en un enano al llegar a Brobdingnag. "
>
> *Uno y el Universo*, Ernesto Sábato

IN MATE, the Analyzer module manages the performance analysis phase and has the logic to require instrumenting and tuning of the application. Until now, Analyzer has been working in a centralized manner, which entails some problems as the amount of processes (and in consequence the volume of events) involved in the application increases. The event collection and the evaluation of the performance model may cause a bottleneck in MATE. This limits the usability of MATE, due to it does not exhibit scalability properties. In this chapter, we propose a new approach to overcome the limitations of Analyzer and make MATE scalable. We start by emphasizing the more relevant aspects of centralized Analyzer, what are the motivations to provide Analyzer with scalability properties and what are the possibilities to reach such objective. Then, we present our proposal, the *distributed collection and hierarchical preprocessing* of data and we analyze the results of applying the new approach. Finally, we study the overhead caused by the use of MATE.

3.0.1 Centralized Analysis Approach

In MATE, the performance analysis is originally done in a centralized manner. Analyzer is executed on an independent machine to reduce the overhead caused by the continuous analysis process in machines where the application is running. This is shown in figure 3.1.

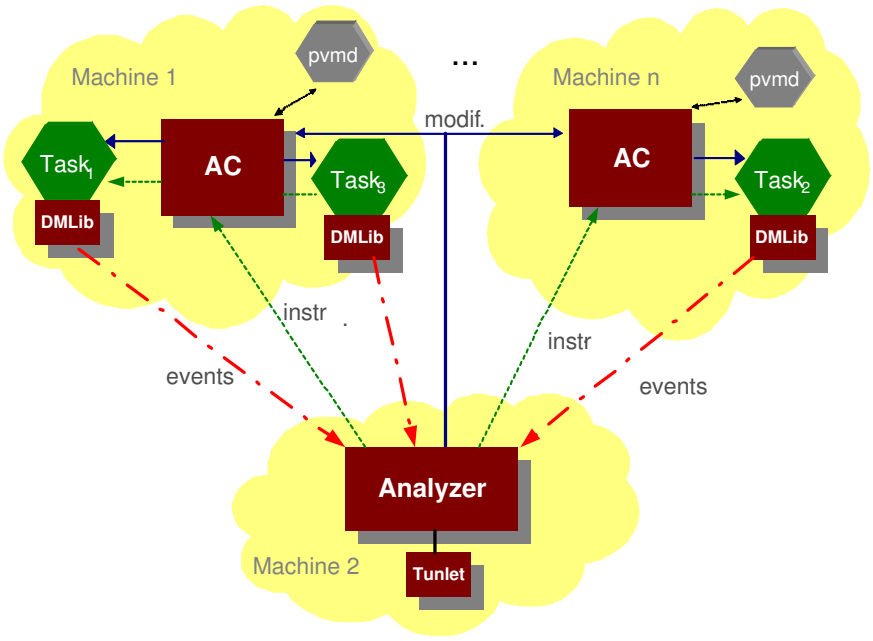

Figure 3.1: Analyzer interacting with the rest of the environment

As explained in chapter 2, the Analyzer module provides the Monitor module with the measure points. These points indicate the instrumentation which has to be inserted into the application in order to collect information on the behaviour of the application (in Figure 3.1, dashed arrows). Collected information related to each task of the application is sent back to the Analyzer as events (dashed-dotted arrows in the figure). When all the information is collected, Analyzer can effect the performance analysis of the application, by evaluating the performance model. Depending on the result of the evaluation, Analyzer decides if it is necessary to introduce some changes into the application to adjust its behaviour to the current conditions of the execution environment. If so, the Tuner module of AC applies the required changes in the application (illustrated by continuous arrows).

All the knowledge about a particular performance problem, i.e. measure points and performance functions, is condensed into a tunlet. While the performance analysis is being executed, the application continues its execution. When a tuning action is required, the execution of the application is stopped for a while, the tuning action is performed and then the execution is resumed. Tuning actions could require synchronizing the tuning action, then it will be necessary an additional stopping in order to insert a breakpoint.

3.0.2 Scalability of MATE: Motivation

At this point, the following aspects of the parallel paradigm should be considered:

- The number of machines involved in the execution of the application
- The persistence of the performance problems

With regard to the first point, we can assume that as the number of machines increases, the number of events incoming to the Analyzer rises too. As a consequence, the Analyzer is turned into a bottleneck which affects the global effectiveness of the system. The operation of the Analyzer is shown in more detail in Figure 3.2. There is a thread collecting the events incoming from the application. Each event is managed to extract the information associated to it; such information is used to calculate or update the value of the intermediate auxiliary variables used to store the information until every event of the iteration is processed, and all the information is available to calculate or update the performance parameters.

Once every parameter is determined, the performance model is evaluated to decide if some change is necessary to improve the behaviour of the application. As the figure illustrates, independently from the amount of incoming events, they are managed following a *first-in, first-out* policy. It provokes the time wasted in processing the information is proportional to the amount of events. In addition, sometimes there are "waves" of events; this means that every task is sending events at more or less the same time, causing the overloading of the Analyzer in a particular instant, such as the end of an iteration, when every process in the application ends. Even more, while the

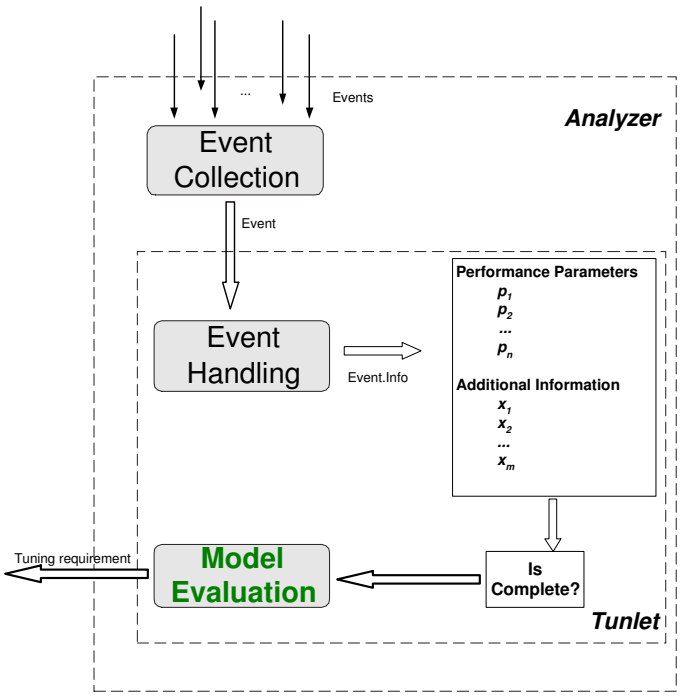

Figure 3.2: Analyzer operation

Analyzer is processing data for iteration i the application follows executing iteration $i + 1$, thus the collection of events is continuously in operation.

Taking into account the second point, we should bear in mind that the application tuning is based on the assumption that performance problems last more than one iteration. That is why the evaluation of performance models should be reduced to evaluate a set of expressions. However, in order to evaluate the expressions it is necessary to process all the incoming events to obtain the values of the model parameters. Similarly, when the number of events goes up, the processing of the information associated to them takes more time. If we consider these two situations at the same time, the bottleneck caused by collecting events and their subsequent processing could mean that when the solution to the existing problem is ready to be inserted, perhaps the performance problem has changed or disappeared. Figure 3.3 illustrates an example of this delay in making effective the adaptation of the application.

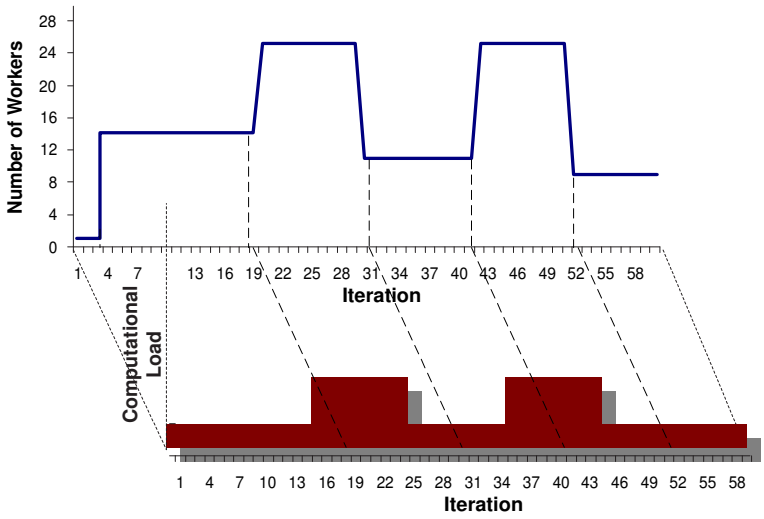

Figure 3.3: What happens when the Analyzer is overloaded

In the example of the figure, a Master/Worker application which executes through 60 iterations, is subdued to a controlled extra load. Certain variable load is injected in the system to provoke variations in the current conditions and the consequent reactions of MATE to adapt the application to the new conditions. In this case we consider a tunlet to tune the number of workers. As can be seen, the number of workers is adapted as the load pattern changes. As the load in the system increases, the number of workers is changed into a bigger one. Conversely, the number of workers is reduced as the load in the system decreases. However, the modifications in the number of workers is delayed several iterations due to the amount of workers and events involved in the obtaining of the performance parameters. In consequence, the first iterations in which the load in the system increases are penalized due to every worker is overloaded. Once the number of workers is tuned the performance of the application improves. However, as the load in the system decreases the first iterations are penalized by the excessive amount of workers, due to each worker can process the task faster. Thus, along the execution of the application there is a continuous lag among the conditions in the system and the tuning actions.

In this context, we need to determine a means of executing the analysis phase in a more useful manner. Then, the scalability of Analyzer arises, and

it is necessary to study what strategies can be used to implement it to adapt the application as fast as possible.

Before studying the possible approaches to overcome the problems presented by centralized Analyzer, we have to consider what the requirements of a viable and useful option are:

- to exhibit scalability properties
- to cover the delays in collecting and processing the events in order to apply the tuning actions in a more sensible manner
- to be an automatizable approach. In other words, as introduced in Chapter 1, the aim of this work is to automate the tuning process as much as possible. In Chapter 4 we will present a methodology to automatically insert performance knowledge in MATE. Such knowledge is used in order to manage the monitoring, analysis and tuning phases, due to it includes the measure points, the performance model and the tuning points/actions. In this chapter, when considering the selected approach to overcome the difficulties of centralized Analyzer we should have in mind that such approach should be viable to deduce its logic from the specification provided by the user.

The requirements for the new approach impose an additional challenge in order to find a viable solution to scale the Analyzer.

3.0.3 Different Approaches to support Scalability

Considering the growth of parallel computing and high performance computing in recent years, it is essential to improve the tools that computer scientists provide to assist the users' work. In the case of MATE, a bottleneck appears when the number of machines involved in the execution of the application is increased and in consequence the number of events to be managed. It is due to the Analyzer module, which is working in a centralized way. Then, we must analyze all the possibilities to make Analyzer scalable and in consequence make MATE scalable too.

There are two different approaches to make Analyzer scalable:

- The hierarchical approach
- The distributed approach

Both approaches present certain inherent difficulties.

If we consider the *hierarchical approach*, at first sight it seems to be the most useful solution for implementing the scalability of Analyzer. Collecting data and performance model evaluation would be divided into smaller pieces of work, which will be managed at each stage of the hierarchy. This would allow for the decentralization of the analysis phase. The problem with this configuration is that it would be necessary to redefine the performance model of the problem the user is trying to overcome. This redefinition would be needed in order to indicate what should be gathered from the application execution at each stage in the hierarchy and how it should be evaluated. This kind of solution is not very useful due to the performance models are general models, independent from the implementation of MATE, i.e. without consider any hierarchy in the evaluation of the performance parameters and expressions. In addition, as mentioned in the requirements of previous section, when users have to develop a tunlet, it is assumed that they are capable of cooperating to write the specification in an effective way; the user needs have a general knowledge of his application and the performance problem it presents. However, if the new hierarchical approach requires the user to think the model in a hierarchical way, the use of MATE will became very restrictive, due to we are considering non-expert users who have not necessarily been involved in defining the performance model and who therefore do not know the details of the model.

Due to the inconvenients of the hierarchical approach, we can think in multiplying the analyzer without considering hierarchy. Then, the *distributed approach* appears, proposing to execute several instances of the Analyzer in parallel, each one dedicated to manage the behaviour of a certain set of machines in the application. Perhaps it seems to be a very direct proposal to reach scalability, but similarly to the previous approach, it presents some difficulties. In this case, in order to apply the performance model separately in each instance of the Analyzer, we have two contrasting options:

- to send the events in a redundant way to each separate Analyzer

- to change the programming model the application follows

The first choice is relating to the fact that, in general, every programming model has more than one kind of processes cooperating to execute the application. In general terms, performance models are defined in function of the programming model as a whole, then to evaluate they it is needed to collect information about diverse processes in the programming model and their interactions. But if we want to divide the set of task in the application in several subsets to be managed by a separated Analyzer, the performance model will not be straightforwardly applicable, unless each subset of task is recursively equivalent to the global programming model, or the model does not need global information. Unfortunately, in general this is not the case. Then, we could mitigate this situation by sending to each instance of the Analyzer the events external to its set of tasks, in order to evaluate the performance functions. Clearly, by doing this we increase communications in the system and perhaps we can introduce some kind of inconsistency, depending on the performance model definition.

This second alternative approach exhibit a clear problem: we cannot force the user to change the implementation of his application to take advantages from the performance model he is providing. Not only these changes would complicate the task of the user, but they also would provoke that the programming pattern modeled in the performance model would not coincide with the new pattern adopted by the application. In addition, we need some centralized evaluator to verify the global behaviour of the application is balanced among the different groups of machines and all of them obey as a whole to the original performance model. In this case we would need the user to define the global application performance model. Furthermore, as the number of distributed Analyzer processes increased, the central Analyzer will suffer the same problem as the full centralized approach. Then, distributed approach does not seem to be very promising.

In addition to the previous considerations, we are trying to make the tuning process as automatic as possible. As the architecture of Analyzer gets more complex, the involvement of the user in programming its logic will be required. As mentioned in the Chapter 1 the main contribution of

this work is to provide a methodology to automatically insert new tunlets in MATE -presented in Chapter 4-. Thus, the approach followed to decentralize the Analyzer has to consider the viability of deducing its logic from the specification. This adds another constraint to implement the Analyzer in a distributed or hierarchical way, due to deducing an entire hierarchical or distributed performance model from the original performance model could be a very complex task, and hardly automatizable.

Analysing the problems that the hierarchical and distributed approaches present, we propose an alternative approach, to take advantage of the good characteristics of each one. We will distribute what can be distributed independently from the performance model and we will define a hierarchy considering the parts of the model which can be decentralized, considering if such definition is specification-inferred. We call this approach the *Distributed-Hierarchical Collecting-Preprocessing approach*, which will intent to take the maximum profit of hierarchical and distributed architectures, and from the performance model provided by the user, avoiding bottlenecks in receiving events and delays in applying solutions.

3.1 Distributed-Hierarchical Approach

Considering the advantages and disadvantages of each approach discussed in the previous section, we need to choose the best option to scale the analysis process without additional user's effort. Then, we essentially propose a mixture between distributed and hierarchical approaches, taking the best characteristics of each one of them.

As explained in the previous section, the distribution of the entire Analyzer is constrained by the performance model, the application, and the user. The functionality of the Analyzer in a hierarchical way is restricted by the performance model and the user. In other words, we cannot force the user to modify the application or the performance model to use the Analyzer in a distributed or hierarchical way. That is why we propose to distribute what can be distributed without generate additional work for the user, and in this way to maintain the MATE's transparency.

As mentioned before, the bottleneck of Analyzer is caused by the col-

lection of events and their consequent processing and classification as the number of managed events increases. The performance model has to be evaluated in a centralized way because in general the evaluated models take into account a global view of the application performance.

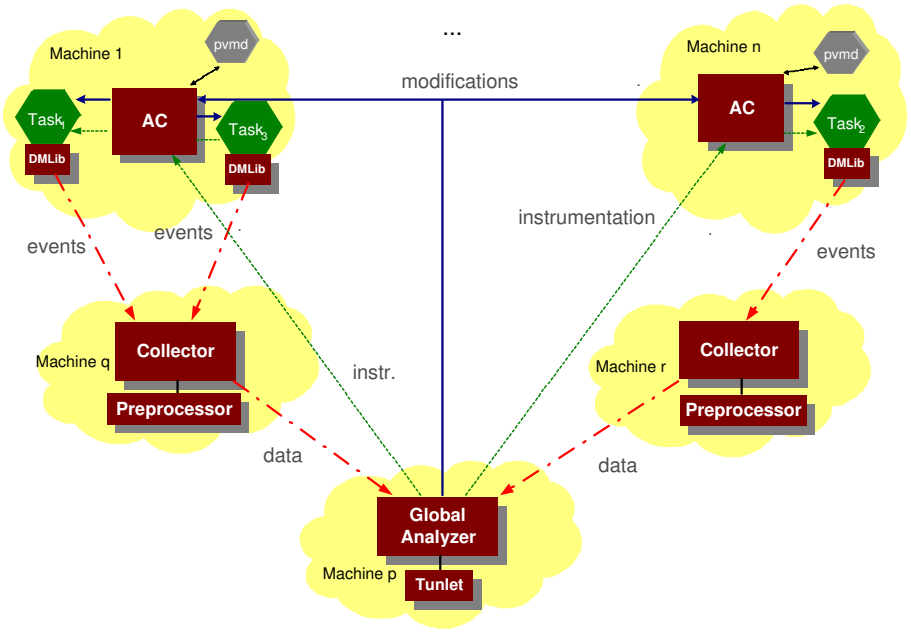

Figure 3.4: Distributed-Hierarchical Collecting-Preprocessing Approach

In our approach, we can distribute the collection of events due to it is independent from the performance model and the application. On the other hand we can also define a basic hierarchy in the processing of the incoming events, when the kind of expression to be evaluated is cumulative or comparative. We only consider these kinds of operations due to their division in hierarchical parts can be automatized without introducing alterations or inconsistencies according to the original operations. More complex operators could be considered, but the involvement of the user should be required when defining the specification, in order to ensure the hierarchical calculations preserves the original meaning of the expression.

In summary, to make the Analyzer scalable we propose to distribute the *collecting and preprocessing* of the information associated to incoming events from the application, which is illustrated in Figure 3.4.

The parallelization of the Analyzer as a Distributed-Hierarchical collecting and preprocessing system, takes the advantages from each one of the subjacent approaches. Then, from this point we can talk about two different kinds of processes cooperating to perform the performance analysis:

- **Collector-Preprocessor**
- **Global Analyzer**

Every Collector-Preprocessor (**CP**) will be an instance of *collection and preprocessing* of incoming data, whilst the Global Analyzer will execute the global analysis according to the information collected and classified by CPs. Both kinds of processes are described in the following sections.

3.1.1 Global Analyzer

The Global Analyzer is the part of the Analysis phase which does effective the evaluation of the performance model. Not only this process evaluates the behaviour of the application, but it also needs to reclassify the incoming information preprocessed in the CPs. This fact is due to the new configuration of the Analyzer. In the first version of MATE, centralized Analyzer received all the incoming events straightforward from the application, more specifically from the DMLibs associated to every task. The events were classified according to their type (i.e., among the different events to be caught -such as start of an iteration, entry to a certain function, etc.- what kind of event it is) and associated information stored in their attributes was used in order to (*directly* when some event attribute embodies the value of a performance parameter or *indirectly* when the event attribute has to be temporally stored until all the information necessary to evaluate the performance parameter is available) update the values of parameters in the performance model.

In the novel conception of Analyzer, Global Analyzer is receiving just a portion of the events, enough to do not overload it with the reception of events rather than with the reception of condensed information from CPs. In addition it is receiving from each CP the condensed information collected from the incoming events of the application and preprocessed to summarize the data needed by the Global Analyzer. Thus, the Global Analyzer is not

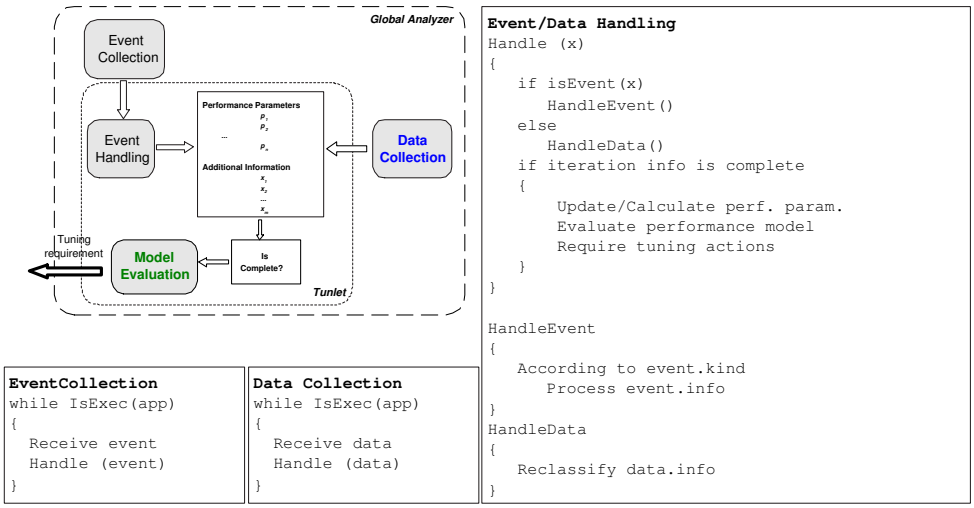

Figure 3.5: Algorithm followed by Global Analyzer

receiving the total set of data collected for every CP, but it is receiving a register containing information to contrast with information received from other CPs, and information condensed about the tasks the CP is responsible for, such as partial additions or products, minimal elements, conjunctions, etc. This is shown in figure 3.5.

3.1.2 Collector-Preprocessor - CP

Each particular instance of CP manages a specific set of machines or tasks in the application. It is responsible for collecting the events of the set of machines associated to it. Not only the CP collects the events, but it also preprocesses the information associated to them. In other words, one of the reasons why we propose to decentralize the Analyzer is to alleviate Global Analyzer from processing all the incoming information. Then, it is needed to implement some logic to preprocess the incoming data in the CP.

To illustrate what we mean by *preprocessing* we can consider, for example, at global level the calculus of average processing time in a Master/Worker application. In order to calculate the average we first need to calculate the cumulative addition of every individual worker processing time. But each CP is managing the events associated to a specific set of workers. Then, no one of them can calculate the total cumulative addition. In order to avoid

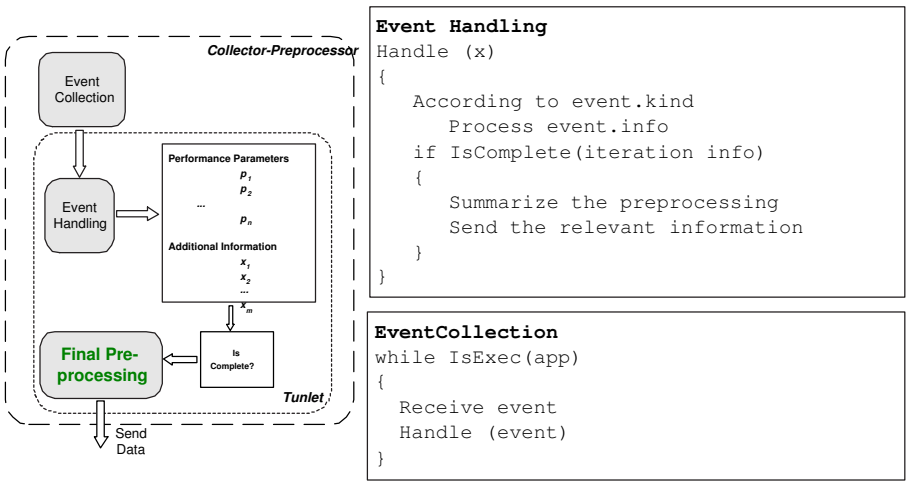

Figure 3.6: Algorithm followed by Collector-Preprocessor

sending the processing times of every worker, each CP calculates the partial cumulative addition of the processing time of the workers it is responsible for. The condensed information is sent to the global Analyzer which will interpret it to obtain all the values needed to evaluate the performance model. In the example, Global Analyzer should add all the partial additions received from all the CPs, and then calculate the average. The general functioning of CP is presented in figure 3.6

Two different aspects raise from the new conception of Analyzer:

- changes in the internal functionality of Analyzer
- changes in the logic to obtain the performance parameters in order to evaluate the performance model.

With respect to both aspects, CPs have to include the mechanisms to manage, classify and resend the incoming data, and the Global Analyzer has to include the logic to interpret the new manner in which the information is arriving. First of all, both CPs and Global Analyzer have to be conscious about the data structure necessary to condense and interchange the information necessary to evaluate the performance parameters. Then, each CP has to be capable of collecting events as in the old-fashion of MATE. However, in place of using the information carried by the events to configure the va-

lues of the performance parameters, the information has to be as processed as possible taking into account the local available information, such as in the example presented in the previous paragraph. The information has to be stored in the common data structure and sent to the Global Analyzer. At the same time, the Global Analyzer has to be capable of collecting and reinterpreting the condensed information incoming from CPs. All this logic has to be introduced in both modules.

These aspects have been considered when distributed approach and hierarchical approach were discarded and distributed-hierarchical collecting-preprocessing approach was proposed: The new configuration of the Analyzer and CPs can be automatically deduced from a tunlet specification without any additional effort of the user. This will be discussed in more depth in the next Chapter, in Section 4.5.5.

3.1.3 Validation of hierarchical-distributed approach

In order to evaluate the effectiveness of the new hierarchical-distributed approach, in the following we present some experimental results, comparing the new approach with the centralized approach. In particular, we studied and compared the amount of time each approach wastes in collecting and process the incoming information to be able of evaluating the performance model. The aim of the experiment is to verify if the proposed approach is overcoming the bottlenecks presented by the centralized approach, and thus, we will be able to appreciate the obtained benefits.

To conduct the experiments, we used a 2D N-Body implementation. The application was developed using the Master/Worker framework presented in Chapter 2. Experiments were conducted on a homogeneous cluster.

The configuration was the following:

- *Processor PENTIUM IV 3.0 Ghz*
- *1 GB DDR-SDRAM 400 Mhz*
- *Ethernet card Broadcom NetXtreme Gigabit*

Furthermore, the operating system installed was Fedora Core 4. All the machines were configured to use NFS (Network File System) based on one

server which has the same characteristics as the cluster machines.

The experiments were made considering the problem of the number of workers in the Master/Worker model. The general idea is on the one hand that when there are many workers the communications saturate the system, and on the other hand, when there are not enough workers the master becomes idle and the workers are overloaded. Thus, what MATE tunes in the application is the number of workers according to the current conditions of the system. The model implemented by the tunlet can be obtained from [14], and will be explained in Chapter 5. The amount of workers involved in the experiments varied among 1 and 25. For each worker we monitored 6 different events and we used two CPs in order to collect such events and the associated information.

Since we need to control the load of the system to reproduce the experiments many times, we created certain load patterns -such as in figure 3.3, so that we can introduce and modify certain external load to simulate the system's timesharing. We have conducted our experiments in two different scenarios:

- In the first scenario the application was executed under MATE considering the centralized approach. One separate machine of the cluster was dedicated to run the analyzer.

- In the second scenario the application was executed under MATE but following a distributed-hierarchical approach. One separate machine of the cluster was dedicated to run the Global Analyzer, and a separate set of machines was dedicated to execute CPs.

In both scenarios, each worker was executed in an individual machine.

The main result we obtained from experimentation is the verification of our proposal: the volume of incoming events and data can be managed in a more effective way, which allows for a more effective use of dynamic tuning too, due to MATE can react faster to the changes in the environment.

In figure 3.7 we represented the operation of the old fashioned Analyzer. It handles the events according to their arrival order and when every parameter in the performance model has been calculated using the information

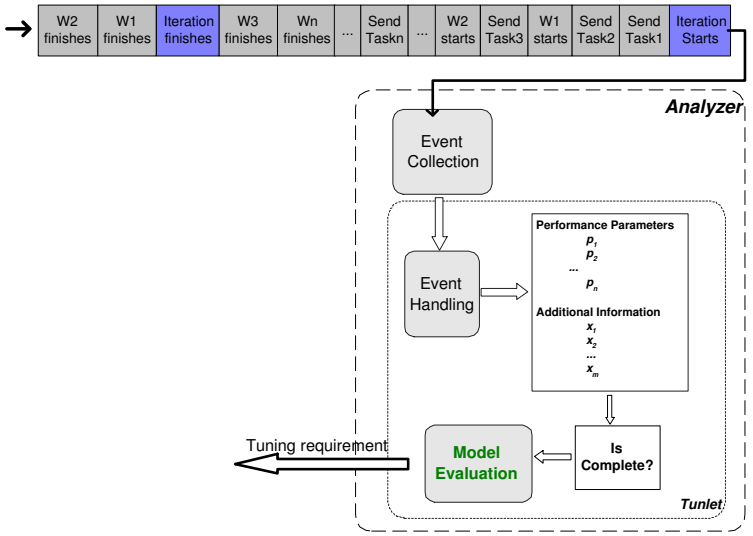

Figure 3.7: Sequence of events managed by the Centralized Analyzer

carried by the events, the model is evaluated. In the event queue, the order of the events cannot be predicted, due to each machine could be working at different speeds and can found different communication traffic. In the figure, even though the $worker_1$ started its task before the $worker_n$, the $worker_n$ finishes before. Specially interesting are the events indicating the start and the end of the iteration (the blue ones), due to they together mean all the processing in the application associated to such iteration has already finished. Thus, the performance model can be evaluated as soon as every event of the iteration is received and handled. In the figure we represented a typical situation when the amount of events is high: due to the volume of events, several events are received after the reception of the event which indicates the finalization of the iteration. This fact provokes a delay in the evaluation or updating of the performance parameters, due to the Analyzer has to process all the information associated to the events before calculating the values of the parameters to evaluate the performance model.

In Figure 3.8 we present the new proposed approach to handle the events. The Global Analyzer handles just a part of the events, while the mass of events is collected and processed by the Collector/Preprocessors. In the figure we illustrated just one Collector-Preprocessor, but in fact several of them

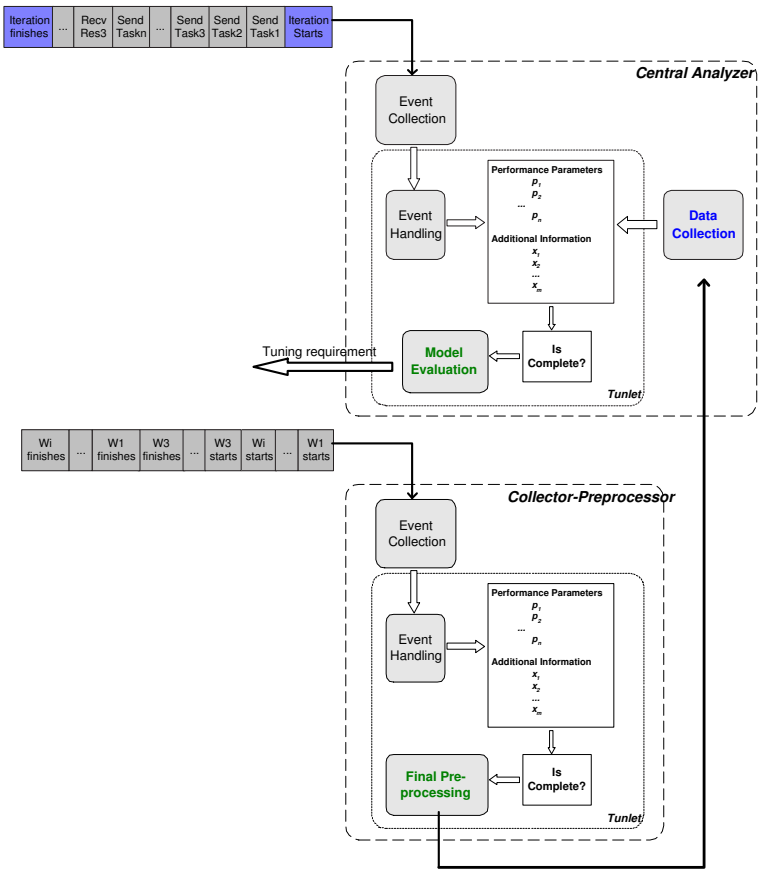

Figure 3.8: Events managed by the Distributed/Hierarchical Analyzer

could be in parallel execution. Once the CP receives all the events and extracts the information, the final calculations to summarize the information are made, and that relevant information necessary for the Global Analyzer is sent. The Global Analyzer assimilates such data into the performance parameters and evaluates the model. The benefit obtained from the use of this approach, is that each CP is handling just a reasonable amount of events to process, and thus the information is classified and preprocessed in an effective and globally faster manner. From experiments, we detected that using this approach, the model can be evaluated when the Global Analyzer receives the event which indicates the finalization of the iteration. In terms of time, this means that the Analyzer is highly synchronized with the execution of the application. In addition, due to CPs send the information preprocessed, the

calculations made to update some performance parameters is reduced; this reduces even more the period of time passed between the instant in which the performance parameters can be calculated and the moment in which the performance model can be evaluated.

In Figure 3.9 we present a general "trace" of the execution of the Analyzer process along an iteration of the application. At left, we represent the order in the tracing, considering the order in which the sentences are executed. Analyzer receives events from the application, among which we emphasized the events to indicate the begining and the end of the iteration (coloured blue), and/or data from CPs (note that *data* from CPs is just received in the new proposal). When every event was received, the performance parameters are calculated or updated and then the performance model is evaluated. Finally, the tuning actions are required.

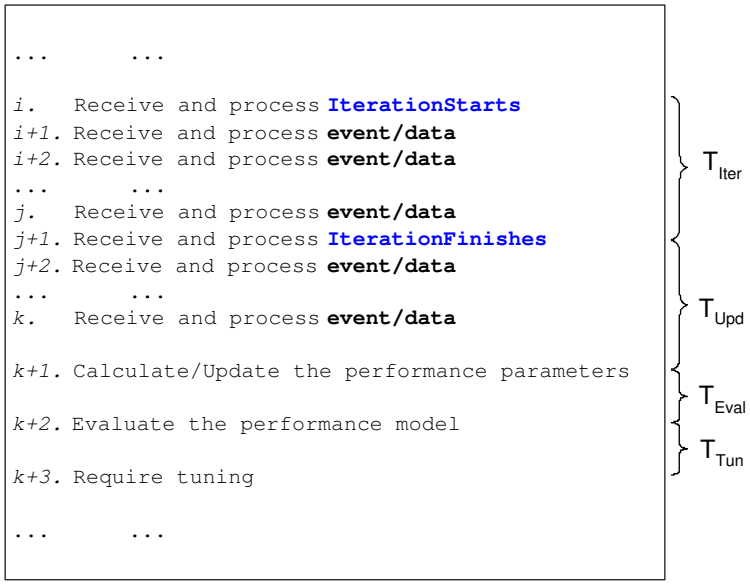

Figure 3.9: Trace of the Analyzer for a given iteration

We studied the following times:

- The time elapsed from the start and the end of the iteration (T_{Iter}, in this case given by $T(j+1) - T(i)$). This allows for calculation of how fast the Analyzer detects the end of the iteration.
- The time elapsed from the end of the iteration and the instant in which

the performance parameters can be updated or calculated (T_{Upd}, in this case given by $T(k+1) - T(j+1)$). This is when every event of the iteration was handled and the information was classified to be available in the updating of the performance parameters.

- The time elapsed from the start of the parameters updating and the instant in which the performance model can be evaluated (T_{Eval}, in this case given by $T(k+2) - T(k+1)$).
- The time elapsed from the start of the model evaluation (T_{Eval}) and the instant in which the tuning actions can be required (T_{Tun}, in this case given by $T(k+3) - T(k+2)$)

Approach	T_{Iter}	T_{Upd}	T_{Eval}	T_{Tun}
Centralized	2199,1	19,7286	3,27	0,2187
Distributed/Hierarchical	680,1	0,2535	2,6	0,2212

Table 3.1: Times obtained for both approaches, in *ms*.

In Table 3.1 we summarize the obtained results, which are illustrated in Figures 3.10 and 3.11.

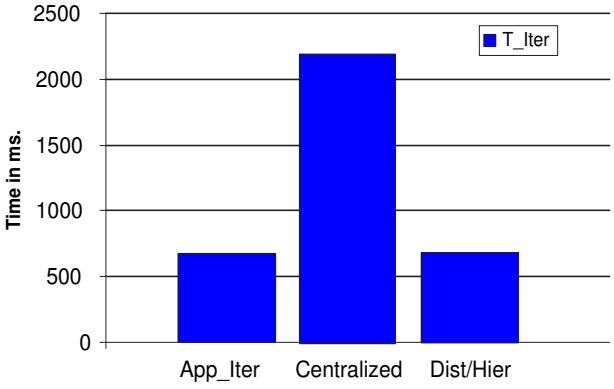

Figure 3.10: Comparison among real iteration time and the obtained for each approach

Note that T_{Eval} and T_{Tun} depend in addition on the complexity of the performance model. Using the new approach, the end of the iteration is detected almost immediately, due to the average iteration time is 672,5 ms (see the Figure 3.10, where the *App_ Iter* bar represents the real iteration

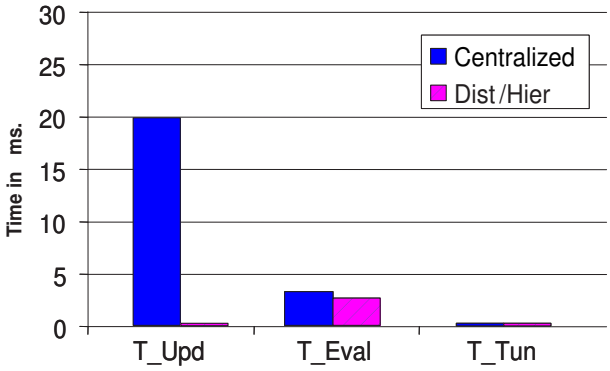

Figure 3.11: Comparison among times obtained for each approach

time). This is a good result, since the Analyzer is more synchronized with the application, which increases the possibilities of tuning the application when the performance problem still exists. Other important result, perhaps the more significant one, is the reduction of T_{Upd} 98,715 per cent, due to all the information is available and classified when the *IterationFinishes* event is received, thanks to the distribution of collecting and preprocessing of the events. In addition, we scaled down T_{Eval} 20,489 per cent. In this case, the percentage of the improvement depends on the amount of updating operations involving in the data preprocessing done by CPs. However, T_{Tun} cannot be reduced due to the performance model is evaluated in function of performance parameters, whose values are already available at the beginning of the model evaluation (instant $k + 3$).

From the obtained results we can propose this new approach to implement the performance analysis in a more effective way, supporting the growing of the applications and the total volume of events.

3.1.4 How to decide the number of CPs?

A key question in using distributed-hierarchical approach is the amount of subordinated Analyzers necessary to effectively overcome the bottleneck suffered by the full centralized Analyzer. From the experimentation, we deduced the ability or capacity of a particular CP is limited by a certain number of events to be received. Unfortunately, in SPMD applications, every process is sending the same events. The events are captured as the process executes,

at specific points (entries or exits of functions). Thus, even though each machine could execute the process at different speeds (caused by heterogeneity or time-sharing) it is normal to receive "waves" of events, specially when we are talking about events indicating points such as the start or the end of an iteration.

The problem is to decide how many collectors are needed, depending on the volume of incoming events. The trouble is magnified when we consider that we are using an automatic and dynamic tuning approach and/or that the number of events can vary along the iteration. The causes to increase (or decrease) the amount of events can be the following:

- some processes are added to (or removed from) the application
- some instrumentation is inserted into (or removed from) the application

The first point is related to cases such as presented in Section 3.1.3, where the number of workers of the application could vary (increase or decrease) along the execution, according to the evaluation of the application performance in the current conditions of the system.

The second point is related to some performance models, which according to the results of the performance evaluation requires additional information of the application or dispenses with some part of it.

In both cases, the effect is similar due to the global amount of events will change. If such volume is very significant, the re-adjusting of the number of CPs could be required to try to preserve the best performance of the system. However, due to the variation of the volume of events will be dynamic, the number of CPs will result in an additional parameter to be tuned in the whole execution, but in this case at level of MATE. Thus, MATE should provide a tunlet to "auto-tune" its number of CPs according to the current needs (this is, the amount of events to be collected and processed). However, there exist two limitations:

- currently, MATE does not support the interaction among different tunlets. Such characteristic would be necessary to manage, solve and counteract the mutual influence among tunlets. In other words, each

tunlet makes decisions according to a specific performance model; however, the decisions could not be good when the tunlet is not taking into account the decisions made by the remaining tunlets, which could be changing some of the crucial parameters. In our case, suppose the tunlet of the number of CPs decides to add new collectors just when the tunlet of the number of workers decides to remove some workers. This will provoke inconsistencies between both models and in the final behaviour and performance of the whole system. This is why MATE should provide composition of tunlets, by defining super-performance models.

- in order to implement a tunlet a performance model of the number of CPs is necessary. We have not developed such a model yet due to it is out of the scope of this thesis, but the results presented in this work can be used as the initial experience in defining the corresponding model. At the same time, we need some mechanism to avoid that as the amount of machines in the application increases the Global Analyzer becomes a bottleneck, such as in the centralized approach. Then, we propose recursively introduce new layers of CPs, as many as needed to cover the amount of machines is been used to solve the application. This would be another reason to denominate "*Hierarchical*" our approach.

In our experiments, we just decided the number of CPs according to the number of workers required by the tunlet, based on the observed behaviour of previous executions (remind we injected controlled load patterns in order to be able of reproducing the experiments). However, this manner of deciding how many CPs have to be used was selected just to verify the improvements proposed by the distributed-hierarchical approach are viable. Therefore, it is required to automate this aspect, due to the functioning of MATE is based on dynamic conditions rather than on history.

3.2 Overhead caused by MATE

The use of a tool to supervise the application execution has some inherent advantages and drawbacks. As mentioned in Chapter 1, instrumentation and

monitoring (and in our case tuning too) affects the performance characteristics of the parallel application providing a false or altered view of its own performance. As MATE is a monitoring, analysis and tuning environment, we have to analyze not only the improvements obtained in the application performance, but we also have to consider the overhead caused by the use of MATE to adapt the behaviour of the application as well as the amount of additional resources required by MATE. Both aspects are studied in the following paragraphs.

3.2.1 Intrusion

In this section we study the aspect related to the intrusion of MATE. To perform the analysis we divided the intrusion in three different types, depending on the nature of the intrusion.

Instrumentation

The instrumentation allows for insertion of new code in the application at certain points, with the aim of collecting information when the execution of the application passes by such place. Depending on the moment in which the instrumentation is inserted (or removed), we can consider two different cases:

- *Initial instrumentation*: in general, the bigger volume of instrumentation is inserted when the application starts its execution, to start the collection of data as soon as possible. Thus, when the application starts, it enters in a phase of overhead caused by the instrumenting process, and just then continues or resumes the execution. The average time wasted to insert the instrumentation to catch an event is 0,284 ms. Even though the time consumed in initial instrumentation could be considerable, it is only suffered at the start-up of the application and is disguised as the execution of the applications progresses. Remind in general we are considering big applications, which could execute several minutes or hours. Thus, the time taken by initialization (in order of microseconds) is hidden by the total execution time (in order of minutes or hours).

- *Extra instrumentation*: as mentioned before, some performance models requires the addition (or removal) of some instrumentation, due to additional information is necessary (or unnecessary) according to the current conditions of the system and the behaviour of the application. The process of inserting (or removing) some instrumentation takes some time, which such as in the previous case, is hidden has the execution of the application continues. In the particular case of removing instrumentation, the time wasted is made up as in monitoring phase some time is saved in capturing such events. The time used to insert extra instrumentation is the same as before.

Monitoring

The monitoring process consists in detecting the instrumented points in the application to gather the required information and send it as events to the Analysis phase. The overhead provoked by monitoring is in the order of microseconds (the capture and sending of an event takes about 0,844 ms) whilst the benefits obtained from the use of MATE reaches the order of minutes.

Note that the overhead caused by monitoring is -in each iteration of the application- proportional to the amount of events to be caught and the number of times that each event occurs through the iteration. Therefore, we can consider that the monitoring process is which introduces a continuous overhead in the application execution, different from instrumenting and tuning. Instrumenting process in general causes the major overhead just at the start-up of the application and eventually when some additional instrumentation has to be inserted or removed. Tuning process introduces overhead as the conditions of the environment change and some adaptation in the application is necessary; thus, if the conditions are not changing along every iteration not too many tuning actions will be required.

Tuning

The tuning process introduces changes in the application. This could be at variables level or at functions level. The overhead provoked will fundamen-

tally depend on the kind of tuning required. In Table 3.2 we summarize the time required to perform the different tuning actions commented in Section 2.2.2.

Tuning Action	*Time*
Set Variable Value	1,1858
Replace Function	2,0078
Insert Function Call	1,308
Remove Function Call	0,254
Function Parameter Change	0,4007
On time function call	1678,286

Table 3.2: Time wasted in the different tuning actions, in *ms*

Some points or actions in the application can be changed without any synchronization, due to they are used at specific points and are out of inconsistencies through a specific iteration. However, some values can only be changed at certain points of execution to ensure the coherence of the value along the iteration. The time wasted when a breakpoint has to be inserted before applying the tuning action is 1,539 *seconds* in average. Just for providing an example, we can consider the variable used in a Master/Worker application to control the current number of workers (named "*nw*"). Supose that the master process uses *nw* as follows:

```
//Master process
main()
{ ...
  nw=initial amount of workers
  while(there are data to process)
  { ...
    divide the total data into nw tasks
    for(i=0;i<nw;i++)
        send 1 task to worker i
    for(i=0;i<nw;i++)
        receive answer
    put together the answers
    ...
  }
  ...
}
```

"nw" is used to decide the amount of parts in which the total work will be divided and how many *sending* and *receptions* will be executed. Clearly, the value of *nw* must be the same along the iteration to avoid anomalies in the execution. Supose, for instance, the following situations:

1. initially *nw=16*, then the master splits the work in 16 tasks and sends them to the 16 workers
2. the value of *nw* is changed into 12 according to the evaluation of the behaviour of the application in the previous iteration
3. the master waits for 12 answers

or

1. initially *nw=16*, then the master splits the work in 16 tasks and send them to the 16 workers
2. the value of *nw* is changed into 20 according to the evaluation of the behaviour of the application in the previous iteration
3. the master waits for 20 answers

In both cases, there are no coherence among the value of *nw* considered at the different points of the iteration. Such incoherences provoke an abnormal or unexpected behaviour in the application, whose consequences could be disastrous. On the one hand, in the first case, the master process is losing part of the answers, then the final resulting data of the iteration will be incomplete due to the inconsistencies in the considered values of *nw*. On the other hand, in the second case the master process becomes blocked waiting for the answers which never will be received.

In order to avoid any inconsistency, the previous problem can be solved in two ways:

- *by synchronizing the modifications*: in this case, the change of the value of *nw* can be made exactly before the next iteration starts. Then, a breakpoint should be inserted to stop the execution in such a point, then the value is changed and the execution is resumed. In this way, the value of *nw* will be fixed through the iteration.

- *by using an auxiliary variable*: this is perhaps the more pragmatic approach to introduce modifications, due to it does not require an additional stopping and resuming of the execution to insert a breakpoint. However, in this case the cooperation of the user is required in case a new variable has to be added in the application. In the example, we could use an auxiliary variable *nworkers* as follows:

```
//Master process
main()
{ ...
  nworkers=initial amount of workers
  while(there are data to process)
  { ...
   nw=nworkers
    divide the total data into nw tasks
    for(i=0;i<nw;i++)
        send 1 task to worker i
    for(i=0;i<nw;i++)
        receive answer
    put together the answers
    ...
  }
  ...
}
```

  In such case, the tuning point will be *nworkers*, then even though its value is changed along the iteration, the value of *nw* will change just when the next iteration starts.

Tuning with or without synchronization clearly presents advantages and disadvantages related to the involvement of the user and the time wasted in applying the tuning action.

Estimation of the intrusion

In order to summarize what the previous paragraphs explains, in the following we present an expression to estimate the intrusion caused by MATE:

$$T_{Instr}(e_1...e_n) + T_{Mon}(e_1...e_n) + T_{Tun}(app)$$

where $T_{Mon}(e_1..e_n) = \sum T_{Monitor}(e_i) * Occurr(e_i)$. T_{Instr} is the time wasted to insert the instrumentation in the application to catch the corresponding n events; T_{Mon} represents the time wasted to catch and send the events, and T_{Tun} represents the time used to tune the application.

In the particular case of T_{Mon}, the individual time wasted to catch an event has to be multiplied by the amount of times the event takes place. This is due to some events are caught several times -such in a iterative function- along the iteration. Sometimes, the occurrence of some particular events is unknown due to it depends on the execution sequence, i.e. it could depends on conditional sentences.

T_{Tun} is the more uncertain time to be estimated. In other words, we can predict how many times will be wasted in effect the tuning actions when required, but we cannot predict when nor how many times the tuning actions will be required, precisely because it depends on the dynamic conditions of the systems.

3.2.2 Additional Resources

Another aspect to consider when using MATE to tune parallel applications, is the additional amount of resources necessary to support MATE. As explained in Chapter 2 MATE is composed of three different modules: AC, DMLib and Analyzer. AC is distributed along the machines involved in the execution of the application, due to is AC which manages the monitoring and tuning process. DMLib is associated to each task of the application to facilitate the instrumentation and data collection and to register the events. Then, both modules share the machines with the application. However, as mentioned previously in this Chapter, the Analysis process is executed in an independent set of machines, in order to reduce the overhead provoked by MATE on the behaviour of the application.

The question is how many additional machines are needed to the analysis process. On the one hand, Global Analyzer is executed in a particular machine, then we have to add a machine to the pool of machines used by the application. On the other hand, each Collector-Preprocessor is executed in an independent machine. One aspect to consider with CPs is that the amount of them could vary along the execution of the application. Thus, the system should have more available resources than when the application starts its execution. Then, we can calculate the amount of machines involved in the execution and tuning of the application as follows:

$$M(application) + CP(events) + 1$$

where $M()$ represents the number of machines involved in the execution of the application, $CP()$ represents the amount of CPs necessaries to manage the amount of *events*, and 1 stands for the Global Analyzer's machine. Note that in the case of $CP()$ it is not only depending on the volume of events, but it is also depending on the distribution followed by the events to arrive to the analysis phase. All these issues have to be considered to define a performance model of the number of CPs, to be included in MATE when it supports multiple tunlets.

Chapter 4

Automatic Development of Tunlets

> "Todo lenguaje es un alfabeto de símbolos cuyo ejercicio presupone un pasado que los interlocutores comparten; ¿cómo transmitir a los otros el infinito Aleph, que mi temerosa memoria apenas abarca?"
>
> *El Aleph*, Jorge Luis Borges

TUNLETS constitute the core of dynamic and automatic tuning implemented by MATE, in terms of representation of knowledge. Each tunlet condenses the information about a particular performance problem that could affect parallel applications. This knowledge is used to manage the monitoring, analysis and tuning steps along the execution of the application. However, the existence of varied performance problems and the need of making MATE transparent to the user, requires a means to automate the inclusion of knowledge in it. In this chapter, we present a methodology to the definition of tunlets and a tool to automatically generate them. In the following section, we introduce what is the spirit of the automation of tunlets creation. Secondly, we define the abstractions and additional terminology used through the development of tunlets. In third place, we determine a methodology to prepare the abstractions needed to define the tunlet. A simple example is presented after that. Next, we define the Tunlet Specification Language and then we describe the Automatic Generator of tunlets from specifications. Finally, we include a section to condense the main aspects

from the user's point of view.

4.1 Introduction

As mentioned in previous chapters, the use of MATE constitutes a very promising approach, especially when we consider non expert users as well as time-shared or heterogeneous environments. In addition, in Section 2.3 we presented MATE not only as a tuning environment, but also as an environment which provides the possibility of developing applications, due to MATE provides a framework for the semi-automatic development of Master/Worker applications. In such a case, when users decide to develop some Master/Worker application by using the framework, they are automatically accessing to the possibility of executing and tuning the application by using MATE, because of it is provided with some tunlets defined over the framework, i.e. over the entities inherent in the framework code (variables, methods and functions of the framework) independent from the entities of each particular user.

The use of MATE as a development and tuning environment results specially useful for non-expert users, due to they are only involved in defining the inherent functionalities in the problem the application is solving (initialization, data partition, data processing, etc.). The aspects related to communications and tuning are automatically tackled by the framework and MATE respectively (for more details see [11]). This situation could be generalized for additional frameworks for different parallel programming models which MATE could include in the future.

However, until now, unless the user developed its Master/Worker applications by using the framework integrated to MATE and he or she wanted to tune the number of workers [10] the use of MATE has not been straightforward. This fact constitutes a too restrictive usefulness of the tuning tool both at level of programming models and development of applications and performance problems that could be tackled. The user can have the application previously developed, following another programming model, or can prefer to develop the application by using another tool. In addition, there exist a variety of performance problems inherent in each parallel program-

ming model. Each problem must be tackled in MATE by an independent tunlet, i.e. the implementation of a tuning element for each performance problem is needed.

MATE is an environment conceived to assist and simplify the user's work. If we think in a particular user who has his or her own parallel application (from now we generalize as "he", for simplicity). He could be in one of the following situations:

- He needs add a new tunlet to overcome a particular performance problem which has not been included in MATE yet.
- He needs to tune a performance problem included in MATE but this tunlet was implemented according to another application (or framework).

Then, he should implement (or re-implement in the second case) the tunlet. But, if the goal of the environment is make easier and more automatic the user's work, it does not make sense involve the user in the implementation details of MATE to be able of programming his tunlet.

In this chapter we present a methodology for automatically developing tunlets with the aim of providing the users with the possibility of:

- developing their applications without the necessity of using the Master/Worker framework included in MATE, and
- adding new tunlets to overcome different problems

Figure 4.1 represents this proposal from the user's point of view. It summarizes the idea of automating the creation of tunlets. Given a particular parallel application it could present some specific performance problem. If the user knows the mathematical model of the problem or he is capable of develop it, that model constitutes a piece of knowledge that can be included as a tunlet in MATE to automatically tune the application during execution. Then, the user only needs to know his application and the performance model to specify it, then the creation of the tunlet is completely automatic.

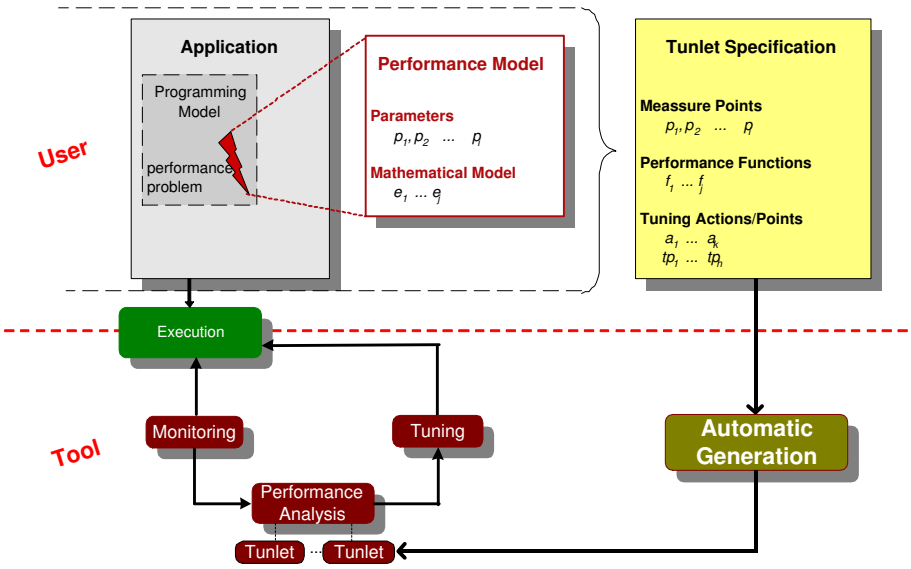

Figure 4.1: Automatic development of a tunlet from a specification

4.2 Abstractions and Terminology

The main objective of this work is to exempt the user from being involved in the implementation details of MATE when defining a tunlet. However it is necessary the user to know MATE in terms of functioning, knowledge, information and structures required to tune the applications. In addition, it is needed the user to know and understand both the application and the performance model in order to develop an effective tunlet. Thus, the user will be able of developing a solution for the problem in a high level of abstraction.

Recaping on Chapter 2, **MATE** *(Monitoring, Analysis and Tuning Environment)* is, as its name indicates, a tool conceived to control and adapt the execution of parallel iterative applications. This environment works in an automatic and dynamic way, characteristics especially useful in both situations: when the user is not an expert on performance analysis and/or when the applications are executing in a heterogeneous or time-sharing environment or their behaviour depend on the input data. The general functioning of MATE is shown in figure 4.2.

As can be seen, MATE works in three different and continuous phases over the application: monitoring, analysis and tuning. When the appli-

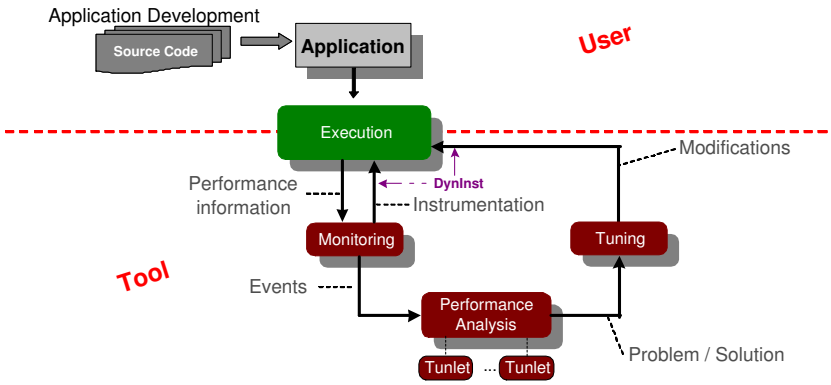

Figure 4.2: General functioning of MATE

cation starts its execution, the monitoring phase inserts some code into -*instruments*- the application in order to collect information about its behaviour along the execution. The gathered information is sent as events to the analysis phase, and is used to evaluate the deployment of the application, by looking for possible bottlenecks. The analysis phase tries to find solutions to overcome such problems, and indicates the tuning phase what is needed to do. Then, the solutions are inserted into the application by the tuning phase, dynamically changing some values or code. This process is repeated along the execution of the application and thus it is adapted to the changing conditions presented in each particular iteration. The insertions/removals or changes made in the application during run-time, are possible because of the use of a dynamic instrumentation technique.

In this process, the user is only involved in the development of the application. However, he needs to know what kind of problems MATE is capable of tackle, in order to know if MATE is useful for his application. In general, MATE can solve every problem which can be expressed by means of a performance model. Performance models constitute the knowledge used by MATE to conduct what information is needed to collect during the execution (so called *measure points*), how to evaluate the collected information (the *performance functions*) and which are the changes needed to tune the application (the *tuning points*). Each performance model is encapsulated in a piece of software, the so called "tunlet".

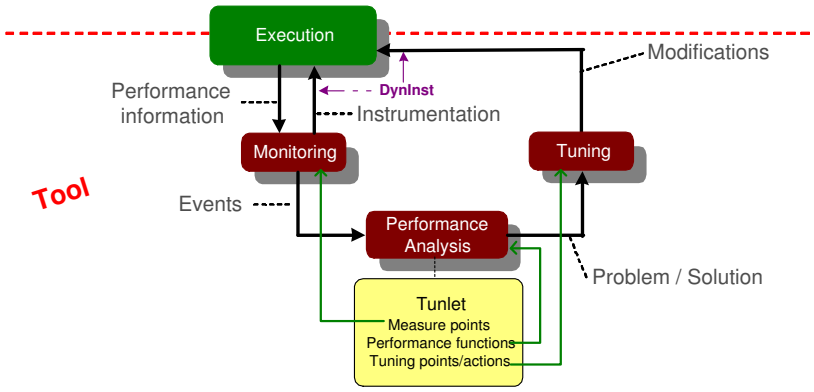

Figure 4.3: Role of a tunlet in MATE

As shows the figure 4.3, the tunlets are part of the analysis phase, and are in fact which own the knowledge. According to what mentioned in the previous paragraph and introduced in Chapter 2, the main elements constituting a tunlet and used by MATE to configure each of its monitoring, analysis and tuning phases (see the green arrows) are the following:

- **Measure Points**, which indicate *what* is needed to measure in the application to be able of evaluating its behaviour. This definition includes values of variables, parameters, function returning, timestamps, etc. According to the measure points of the tunlet, the Monitoring phase inserts the corresponding instrumentation in the application in order to be able of collecting the information needed by the Analysis phase.

- **Performance Functions**, which determine *how* to evaluate the collected information in order to detect bottlenecks. Once the Monitoring phase sends to the Analysis phase all the information required for an specific iteration, the Analysis phase can evaluate the performance functions and decide if some change in the application is needed to adapt its behaviour.

- **Tuning Points/Actions** indicating *what*, *where* and *when* to change in the application execution with the aim of adapting its behaviour. The Tuning phase makes effective the changes in the application taking

into account the point/s or function/s to change, the place in the code and the instant during the execution where the changes should be introduced.

These three elements are defined by considering mainly the performance model of the problem and the application. Then, we define the main abstractions on them:

- **Performance Problem**: Each programming model -such as Master/Worker, Pipeline, etc.- provide a different functionality, characteristics and possibilities of interactions and communication. The execution of applications is directly influenced by such characteristics, both the advantages and the drawbacks. A performance problem constitutes some problem in the application, related to the deployment; i.e., the performance obtained by the application is not according to the expectations. Performance problems can be caused by different reasons, but in general, for every parallel programming model, fortunately there exist a set of well-known performance problems.

- **Performance Model**: In general, in order to obtain a suitable behaviour certain parameters of the applications must be tuned. But the trouble is that some parameters can't be statically tuned due to their dependency on particular conditions of each execution, inherent in the application or to the execution environment. However, parallel applications generally act in accordance with different programming schemes. By studying both the benefits and bottlenecks of these schemes, the performance problems they present can be mathematically modeled. Thus, a performance model is the mathematical model of a particular performance problem. These models can be a means to help us overcome the difficulties in reaching an optimal performance. In general, a performance model is defined by the following elements:

  Performance Parameters: parameters needed to evaluate some expressions to represent the behaviour of the model; are the "mathematical" variables involved in the evaluation of the performance functions.

Performance Functions: are functions expressing how to evaluate the performance parameters. In general, the results of the performance functions are used to determine what the solution to the existing problem is.

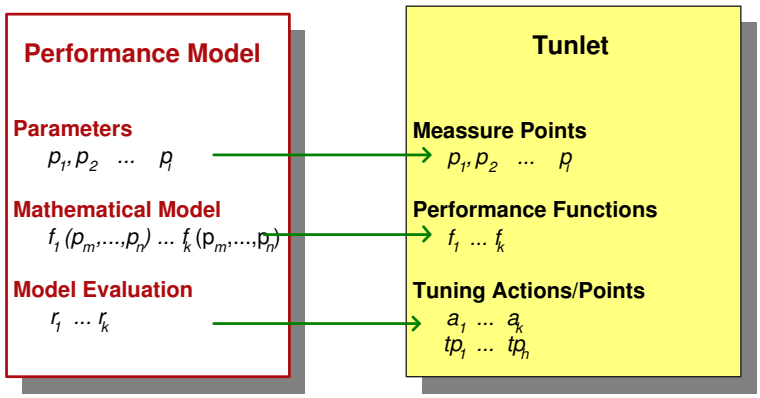

Figure 4.4: Relation between a performance model and a tunlet

The elements of a particular performance model constitute the knowledge condensed in a tunlet, as shows Figure 4.4.

- **Actor**: In general, every parallel application has different kinds of processes executing in parallel and cooperating to solve the problem. Each kind of process or task in the programming model, constitute a different actor. For instance, in the Master/Worker model, master and worker are two different actors, and in the Pipeline model each phase represents a different actor. In figure 4.5 is shown this concept and the following ones.

- **Event**: is the mechanism used by MATE in order to collect information. Events are captured in entries or exits of functions and can carry additional information associated with them. For instance, when captured the entry of the processing function it is possible to collect the volume of data to be processed.

- **Variable**: is a variable of the application. A variable can be needed for obtaining its value or change it.

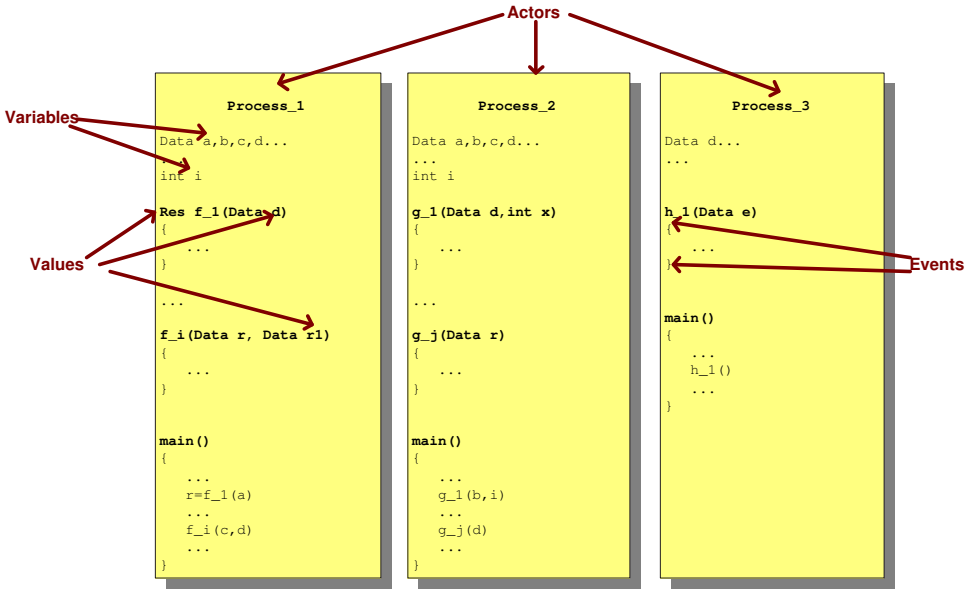

Figure 4.5: Abstractions in the application

- **Value**: is the value assumed by a parameter or the result of a function. Similarly to the variables, values are useful to obtain information or to change such values.

- **Attribute**: is a piece of information or attribute related to a particular entity. Actors and Events are in general the entities which have a set of attributes associated. These attributes condense information related to the entity which is used to determine the value of some performance parameters.

The interdependence among these abstractions can be summarized as shows figure 4.6.

4.3 Methodology

Once defined the abstractions used by MATE and its interface to work over the applications, we present the methodology a user should follow in order to define a tunlet. The methodology includes a series of steps, related to identify or/and interpret the previously defined abstractions in the specific

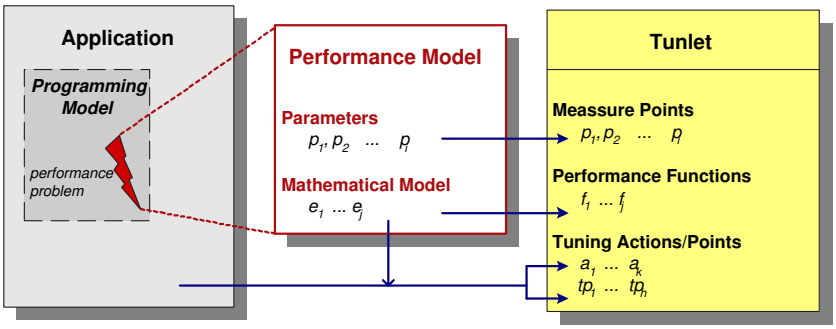

Figure 4.6: Interrelation among application, performance model and tunlet

application and performance model under study.

4.3.1 Providing a performance model

MATE is oriented to non-expert users due to it exempts them from being involved in tuning aspects when developed the applications using the framework included in MATE, and the problem to solve is the number of workers. However, users should have the possibility of inserting new knowledge in MATE with the aim of solving a new particular problem, as well as using the knowledge already included in MATE but applied to other applications.

Given an application with some performance problem, the user has to consider which the model of such a problem is. The performance model can be a preexisting model or can be an *ad hoc* model defined by the user. When the user is involved in developing a performance model, a higher degree of expertise is needed, but we can suppose that if somebody turns to parallel computing has to be conscious about the "collateral" problems the parallel application can present, and has to dominate a certain set of concepts relating to the parallel paradigm, commented in Chapter 1.

4.3.2 Understanding the performance model

Once the performance model was determined, the understanding of it is a basic requirement due to the model has to be interpreted according to the application. Thus, the different elements of the tunlet can be defined. When the performance model is already included as a tunlet in MATE, even

though the user is only going to "adapt" it to its application, the requirement of understanding the performance model is fundamental too, because every part of the model has to be reinterpreted.

Fortunately, in case of using a preexisting model, the user should concentrate specially in the study and comprehension of the relevance and semantic of each performance parameter, rather than in the performance functions due to they could be very complex; as mentioned in Section 4.2, the functions are defined over the parameters; then if the model had been validated and the user has enough understanding of the semantic of each performance parameter to interpret them in the application, the functions will be able of been evaluated.

4.3.3 Interpreting the performance model

This step and the following one are very related due to their interdependency, then they could be done in parallel. When the performance model had been determined and specially the performance parameters have been understood, the user has to determine what are the entities in the application which embody each performance parameter, i.e. how to provide each parameter with its semantics. Here is where the user has to define the events to be captured and the information associated to them.

Identifying the information/variables/values

As mentioned before, the performance parameters have to be interpreted according to the variables, values and functions of the application under study. According to the semantic of each performance parameter, the user has to determine how to constitute its value, and what is the event the information has to be associated with. For each variable or value, the user has to provide its name, data type, and the name of the actor which has visibility of it. A special consideration is needed with variables. The variables whose value will be obtained or changed, have to be global variables. Thus, this can require some adaptations in the implementation of the application, redefining variables as global as well as using auxiliary global variables. In Figure 4.7(b) we declare b as global not only to make it visible for $Assign2B$,

but also to be able of obtaining its value and send it as an attribute of the event caught in the exit of the function.

Identifying the events

Events constitute the mechanism of MATE in order to collect information. Then, according to the semantic of each performance parameter (or some group of parameters), the user has to determine what are the entries and exits of functions which have to be caught, i.e. what are the points of the program in which the information has to be obtained. Sometimes, -specially when the required information required is related to timestamps- the user will be forced to abstract some parts of the functionality of the application in functions in order to be able of delimiting the beginning and the end of such an interesting point of execution.

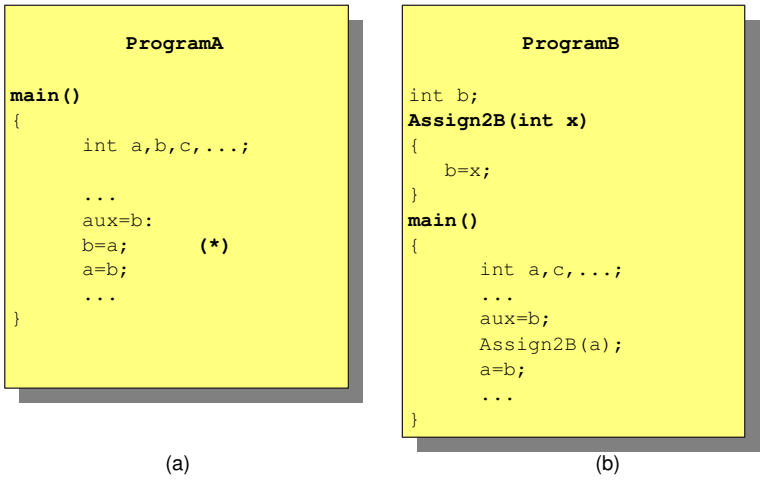

Figure 4.7: Two simple and equivalent programs

For example, in order to caught the instant (*) of Figure 4.7 (a), it is needed to modify the program as in Figure 4.7 (b). In this way, when exiting from function $Assign2B$ we will capture the timestamp of its exit. In general, for each event the user has to determine a name and specify its location in the code (method, class and place) and the actor where it happens.

4.3.4 Identifying the actors in the application

When the performance model to tune the problem presented by the application was determined, it is needed the user to abstract what are the actors involved in the application. For each actor, the user has to provide a name, the *file* and the *class* in which is included or defined, and the name of the *executable file*. The user should reflect over some additional aspects:

- the *minimal* and *maximal quantity* of this type of actor could co-execute in the application.
- a *completion condition* to detect when the actor reached the end of its tasks along an iteration.
- the *actor's attributes*, i.e. what properties should be registered in each iteration; for example for a worker, to catch the computing time along the iteration could be interesting.

4.4 Simple Example

In order to clarify the functioning of the proposed methodology, in this section we present a simple but complete example.

Suppose the parallel application presented in Figure 4.8. The application consists of two processes: *Process_1* and *Process_2*. *Process_1* initializes the data to be processed and partitionates it. Then, on the one hand a half of the data is sent to *Process_2*, which processes and send back the results to *Process_1*. On the other hand the other part of the data is processed by *Process_1*. When *Process_1* receives the results from *Process_2*, it executes the final treatment of the results. All this process is iteratively executed.

Supose the performance of this application do not covers what expected. The user wants to analyze how much time is wasted in communications. Then, the user has to provide the mathematical model of the execution time of an iteration. Note that for simplicity, in this example we talk about mathematical model rather than performance model, due to the study of the time wasted in computing and communications are not enough to improve them.

```
Process_1

Data dt,dt1,dt2,rs,rs1,rs2

Initialize(Data d)
{
   //Initialize data
}
Finalize(Data r, Data r1, Data r2)
{
   //Final processing of results
}

DivideData(Data d, Data d1, Data d2)
{
   //Divide d into d1 and d2
}
SendData(Data d)
{
   //Send d to Process_2
}
Receive(Data r)
{
   //Receive r from Process_2
}
Data Process(Data d)
{
   //Process corresponding Data d and
   //return the results
}

main()
{
    while(!end)
    {
        Initialize(dt)
        DivideData(dt,dt1,dt2)
        SendData(d2)
        rs1=Process(d1)
        Receive(r2)
        Finalize(rs,rs1,rs2)
    }
}
```

```
Process_2

Data dt,rs

SendData(Data d)
{
   //Send d to Process_2
}
Receive(Data r)
{
   //Receive r from Process_2
}
Data Process(Data d)
{
   //Process corresponding Data d and
   //return the results
}

main()
{
    while(!end)
    {
        Receive(dt)
        rs=Process(dt)
        SendData(rs)
    }
}
```

Figure 4.8: Example of a simple parallel application

1. **Providing the model**. We can think how to model the execution time of an iteration in a parallel program. Unlike the sequential paradigm, in the parallel paradigm the data processing is overlapped -or folded- and this provides a shorter total execution time. But in turn, there is an additional time, the communication time spent to coordinate all the parallel processes. Figure 4.9 is a diagram of activities in which this situation can be seen. Dashed lines represent the time line. Every iteration starts by a brief period of initialization. After that, process 1 (P_1) divides the set of data to be processed into two parts. One portion is sent to process 2 (P_2) and the other portion is processed by process 1. When process 2 finishes the reception of the data, starts its computing phase. Following, when all data has been processed, the

results are sent back to the process 1. After a brief phase of finalization of iteration, for example to put together all the results, it starts the next iteration -note that in terms of time, each iteration covers its own period of time, but due to the algorithm is the same for every iteration, we represent this fact with the last arrow which joins the finalization of one iteration with the initialization of the next one-.

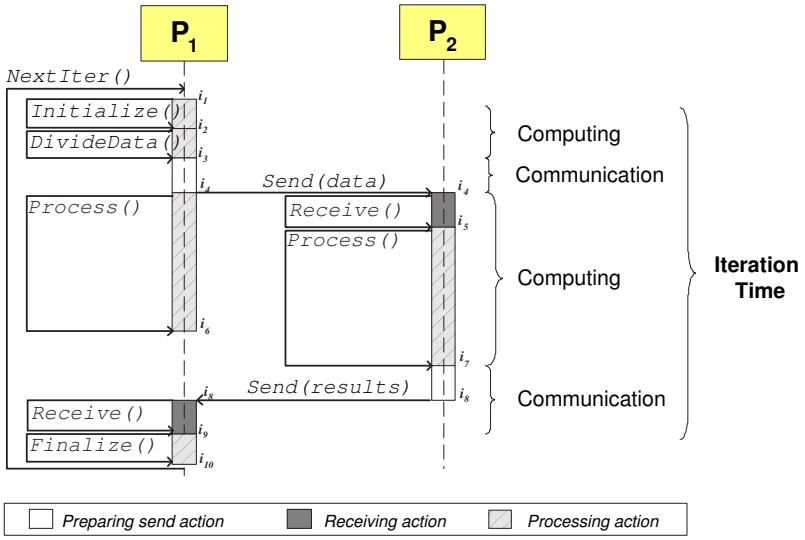

Figure 4.9: General view of a parallel application and the iteration time calculation

As can be seen, by dividing the total work into two parts it is possible to reduce the total execution time in comparison with the time it would take to process the data sequentially, in spite of the additional communication time and the waiting time. For example, P_1 starts and finishes the processing phase before P_2, then it spent a while waiting for the results of P_2 to execute the finalization of the iteration. Sometimes, some of these times are overlapped with computing time which reduces the negative effects of waiting. In a scheme like presented in Figure 4.9, it is possible to determine the execution time of the iteration it as follows:

$$T_{Ex}(it) = T_{Comp}(it) + T_{Comm}(it)$$

where:

$$T_{Comp}(it) = T_{Ini}(it) + T_{Proc}(it) + T_{Fin}(it)$$
$$T_{Comm}(it) = T_{Send}(it) + T_{Recv}(it)$$

In order to calculate each one of the components of T_{Comp} and T_{Comm} we can consider the instants of the entries and exits of different methods or functions.

- T_{Ini} Comprises the initial treatment and configuration, previous to the parallel processing. In our example, it includes the *Initialize* and *DivideData* functions. Similarly, for T_{Fin} we consider the final treatment made over the processed data.
- T_{Proc} comprises the interval of time in which the processes are processing the data.
- T_{Send} comprises the time spent in sending messages, whenever this time is not overlapped to computing time. T_{Recv} bears some resemblances to the previous case, but considering the receiving messages process.

Then, by capturing entries and exits of the correspondent functions we are able to calculate the total time of an iteration. Performance models of programming models are clearly more clomplex than this simple example, but in general they follow the same essence.

Each model should be defined by a set of parameters needed to evaluate the expressions. For the previous example, in spite of it is not a performance model, the mathematical model can be expressed as shows Figure 4.10, where $T_{something}(x)$ represents the time wasted during the iteration x to execute the *something* function, and $Entry(f)$ and $Exit(f)$ allow obtain the initial and final instants of a certain function f, respectively.

2. **Understanding the model**. In this example, due to we followed the reasoning to obtain the model, the semantic of each parameter appears clearly.

Mathematical Model
Iteration Time

Parameters

$Entry$(Initialize())
$Exit$(DivideData())
$Entry$(Finalize())
$Exit$(Finalize())
$Entry$(Send())
$Exit$(Send())
$Entry$(Receive())
$Exit$(Receive())

Mathematical Model

$T_{Ini}(i) = Exit(DivideData()) - Entry(Initialize())$
$T_{Proc}(i) = max(Exit(Processing())) - min(Entry(Processing()))$
$T_{Fin}(i) = Exit(Finalize()) - Entry(Finalize())$
$T_{Send}(i) = max(Exit(Send())) - min(Entry(Send())$ **:**
$[\ \forall\ Entry(Processing()):\ Entry(Send()) < Entry(Processing())] \wedge$
$[\ \forall\ Exit(Processing()):\ Exit(Send()) < Exit(Processing())]$
$T_{Recv}(i) = max(Exit(Recv())) - min(Entry(Recv()))$ **:**
$[\ \forall\ Entry(Processing()):\ Entry(Recv()) > Entry(Processing())] \wedge$
$[\ \forall\ Exit(Processing()):\ Exit(Recv()) > Exit(Processing())]$
$T_{Comp}(i) = T_{Ini}(i) + T_{Proc}(i) + T_{Fin}(i)$
$T_{Comm}(i) = T_{Send}(i) + T_{Recv}(i)$
$T_{Ex}(i) = T_{Comp}(i) + T_{Comm}(i)$

Figure 4.10: Mathematical model of the iteration time calculation

3. **Interpreting the model according to the application**.

   Even though the proposed mathematical model was *had hoc* developed for this simple application, we can determine what are the interesting points of measure in more depth. In order to calculate each one of the components of T_{Comp} and T_{Comm} we have to determine what are the points in the execution which correspond to the instants we need in order to calculate the times. We can consider the instants of the entries and exits of different methods or functions, which are represented in the figure 4.9 as i_x.

   - T_{Ini} can be obtained from the subtraction between the exit of the *DivideData()* function and the entry of the *Initialization()* function ($T_{Ini} = i_3 - i_1$). Similarly, for T_{Fin} we consider *Finalize()* action ($T_{Fin} = i_{10} - i_9$).
   - T_{Proc} is calculated by considering the instant in which the first

process started to process any data (in the example P_1), and the time by the last process finished of processing its data (P_2) ($T_{Proc} = i_7 - i_4$).

- T_{Send} comprises the time spent to sending messages, whenever this time is not overlapped to computing time ($T_{Send} = (i_4 - i_3) + (i_8 - i_7)$). T_{Recv} bears some resemblances to the previous case, but considering the receiving messages process ($T_{Recv} = i_9 - i_8$). Note that interval ($i_5 - i_4$) was not considered as part of T_{Recv} because it is overlapped to T_{Proc}.

Then, by capturing entries and exits of different functions we are able to calculate the total time of an iteration.

4. **Identifying the actors in the application**. In this example it is very clear the existence of two different actors: Process1 and Process2. For each one of them we have to determine some properties -some of them could appear redundant in this simple example-:

   - **Process1**
     - *name:* p1
     - *class:* none
     - *executable file:* Process_1
     - *completion condition:* exit of *Finalize()*
     - *attributes:* timestamps of: *Entry(Initialize)*, *Exit(DivideData)*, Entry and Exit of *Send, Receive, Process* and *Finalize*.
   - Process2
     - *name:* p2
     - *class:* none
     - *executable file:* Process_2
     - *completion condition:* exit of *Send()*
     - *attributes:* Entry and Exit of *Send, Receive* and *Process*

The property *class* is *none*, due to this simple example is written in C (like). If it were written in C++ we will need the name of the class.

4.5 Tunlet Specification Language

As we introduced before, the performance models provide useful information about the deployment of applications. Our goal is to encapsulate all this information on how to solve an specific performance problem as a **tunlet** which will be used through the execution of the application to direct its monitoring, analysis and tuning. In the previous sections we defined a set of abstractions and a methodology to understand the application and the performance model in the terms that MATE does. Once the user reached the conceptual definition of the elements of the tunlet -by following the methodology depicted in Section 4.3- it is necessary a mechanism to formalize such a conceptual definition and transform it into a tunlet. In other words, to define a tunlet, a specification of the performance problem in function of the application needs to be provided. Then we propose a Tunlet Specification Language such that the tunlet can be defined without entering in implementation details of MATE.

According to [2] we defined:

1. The set of symbols which can be used in a valid specification,
2. The set of valid specifications, and
3. The "meaning" of each valid specification.

In the definition of the Tunlet Specification Language there is involved the set of concepts introduced in sections 4.2 and 4.3. However, in this section they are more dependent on the implementation details of MATE, which in several cases determines the way in which the elements are defined. In the particular case of specifying tunlets, it is needed to examine and consider what elements of the performance problem and of the application must be included, how to establish the relationships among them, and how to express its syntax and semantics. All these issues are in relationship with the way in which the Analyzer **represents** and **uses** the knowledge.

Just recapping what explained in Chapter 2, from a functional point of view, the Analyzer is divided into two main parts which are the ***Dynamic Tuning API***(DTAPI) and the ***Tunlets*** [38].

The DTAPI constitutes the interface of the Analyzer to communicate with the Monitoring and Tuning phases. It provides the Analyzer with a global view of the application, the tasks and the events. With regard to ***Tunlets***, they should use Dynamic Tuning API to manage the application by invoking monitoring and tuning requirements, and to handle the events to collect the information of the application necessary for its analysis. Then, the DTAPI constitutes the interface that tunlets must follow to correctly work in MATE. As commented before, descriptive classes of the application are *Application*, *Task*, *Event*, *Attribute* and *EventRecord*. Then, in terms of tunlets specification we should have in mind these classes of the interface and use it as guideline to build tunlets according to the requirements of MATE.

Our study on how to define the Tunlet Specification Language is centered in the following aspects:

- *How to capture the information.* We have to consider the methods provided by the DTAPI in order to instrument the application, in particular, what are the properties which define an event. In order to insert an event in a particular process, it is needed a name for the event, its location in the code and the attributes or information associated to it. Such attributes have to be visible to the process where the event is being inserted.

- *How to manage the collected information.* When an event is inserted in a process, it is necessary to determine a **handler** for such event when it happens and is received by the analysis phase. In that way, the information carried by the event can be obtained and managed as needed. In general, tunlets are the handlers of events, due to they encapsulate the logic to process the information and interpret it according to the performance model.

- *How and where to store the information.* Each tunlet manages a data structure for each iteration. In such structure, the information collected is stored according to the nature of the information: information about actors or iteration.

- *How to modify the application.* Similarly to the insertion of monitor-

ing instrumentation, when some tuning action is required we have to consider the methods provided to introduce some modifications in the application. Thus, we need determine if some synchronization is necessary to introduce the changes, and we have to provide the necessary values according to the kind of requirement.

Having in mind these four points, we need a specification to cover all these characteristics. Before providing a formal definition of the language we will analyze and present the language in a more intuitive way.

4.5.1 Components and Sections of the specification

By automating the creation of tunlets we will make MATE transparent to the user. But we require the cooperation of the user in defining his tunlet. As we mentioned before, a specific performance model constitutes the basis of a tunlet due to it provides the measure points, the performance functions and the tuning points and actions. Then, from the **performance model** point of view it is needed to define:

- The *measure points*,
- The *performance model*, and
- The *tuning actions and points*.

However, to instrument the application to monitor and/or tune it, the performance model by itself is not enough, it is needed some additional knowledge about the application, such as the variables which we will use as metrics, the values we are able to change, and the programming model, among others, to have a conceptual view of the application. Then, we need to determine what is required in order to write the specification. Thus, from the point of view of the **application** we need to be aware of:

- The *programming model* it follows, i.e. how different kinds of processes or *actors* are involved in the scheme,
- The *variables or values* we can manipulate, both to get their values or to change them, and

- The functions whose execution we need to detect to collect the information and send it as *events*.

As we can see, the variables, values and events in the application are closely associated to the measure points of the tunlet, because they constitute the interpretation of the variables in the performance model. Then, we can extend the meaning of "*measure points*" and think about the inclusion of this information joined to the metrics we should obtain. In this way, the specification results divided into three different sections, which we describe in the following paragraphs.

Measure Points

The *measure points* section will embody the larger part of the specification, due to it will condense all the information of the application, the programming model and all the parameters susceptible of change, in addition to the performance model parameters.

The most direct way to describe the application is perhaps by following the classes in the DTAPI. But even though we should have that structure in mind, we must remind our goal: simplify the user's task. Then, we must consider the basic parallel concepts that the non-expert users should manage, such as the parallel programming models, and the results they expect to obtain by using them. In addition, as users are going to specify a tunlet taking into account a certain performance model, it is needed they understand how such a model can act over their applications. Then, different processes are defined as actors of the programming model.

The user must define:

- The **actors** of the programming model (the types of processes or tasks co-existing in parallel). The tunlet needs this information because in general each kind of process must be instrumented in a different and specific sense inherent in its nature and role in the programming model. In other words, for each actor it is needed to capture different events, then the instrumentations inserted to detect them will depend on the necessities. Then, according to DTAPI, when a task is registered, in order to locate the points in the code and discriminate what kind of

instrumentation should be inserted, it is needed to declare the *name* of the actor and the *class* in which is included or defined, and the name of the *executable file*. Some additional information is required from the tunlet point of view:

  - the *minimal* and *maximal quantity* of this type of actor could co-execute is needed to generate the structures to manage the behavioural information of each process along the successives iterations.
  - a *completion condition* to detect when the actor reached the end of its tasks along an iteration. This is necessary to be able of checking if every process finished before evaluating the performance functions.
  - the *actor's attributes*, i.e. the properties should be registered in each iteration; for example for a worker, to catch the computing time along the iteration could be interesting. The attributes are normally used to calculate the value of other attributes, or performance parameters.

- The **variables** and **values** which can be instrumented or tuned in the application. For each one must be declared:

  - the *name*, such as in the application
  - the *data type*, such as declared in the application
  - if it is a *variable*, a *parameter* or a *function output*
  - the *actor* who has visibility of it.

  In general, these variables and values are used in defining the attributes of the specification elements. The details (type, actor, etc.) are needed to locate them in the code and transmit them correctly.

- The **events** to capture, such as entries or exits of functions. Each event is defined by its *name*, the *actor* it is associated with, the *place* in the source code and a code to indicate if the event must be used to control the beginning or the end of the iteration. In addition, the user has to

indicate the *utility* of the event through the execution; i.e. if the event have to be always caught or it is an event which could be required according to the evaluation of the system or if it is a removable event. Some *attributes* -that is some information measured when an event occurs- can be associated to a particular event. The quantity of bytes sent, can be an interesting metrics caught when an event indicating the exit of sending function occurs.

- The **model parameters** are the own attributes of the performance model, whose value generally is calculated as a function of the attributes of actors or events. Even though these parameters could be omitted in the tunlet, due to the performance functions can obtain the values directly from the attributes of the entities, it is convenient to respect the parameters of the performance model to avoid errors in the interpretation.

- As MATE was designed to tune iterative applications, collected information should indicate what iteration corresponds to, because communications could cause a gap between the instant in which the information is sent and the moment in which it is received. Then, to avoid inconsistencies, we require an additional section in the specification to collect information about each iteration. The iteration information, includes an attribute to indicate the current iteration, and then all the additional information necessary to describe the behaviour of each iteration, according to the performance model.

In general, all the elements in the specification with a set of attributes, must declare for each attribute its name, the data type, the initialization value and the way in which its value must be calculated in each iteration. Finally, if the attribute depends on another attribute or event it should be expressed to maintain the coherence in calculating values.

Performance Functions

With respect to the performance functions, they must be defined in C/C++ language, i.e. any function which could be defined in such languages, will

be recognized as a performance function of the tunlet. This function will be the value assigned to some of the tuning points in the following section or in intermediate calculus. Any library needed to implement the functions, should be declared in the *include* section.

Tuning Actions/Points

There are different possible tuning actions to modify the behaviour of the application, explained in Chapter 2: `SetVariableValue`, `ReplaceFunction`, `InsertFunctionCall`, `RemoveFunctionCall`, `OnTimeFuncCall` and `FuncParamChange`. All the information about a tuning action, i.e. *what* to do, *where* and *when*, is encapsulated as a *tuning point*. Therefore, for each tuning point it must be declared the kind of action (one of the previously enumerated), the identifier of the entity to be managed (the name of a variable or a function), the value to be assigned, a condition to apply the tuning, and additional information about synchronization on the appropriate execution place to change the value of the point. In case the tuning action is `InsertFunctionCall` or `OnTimeFuncCall` a list of attributes -the list of the arguments of the function- has to be added to the previous information. In case the action is `FuncParamChange`, the index of the parameter to be changed is required and a boolean value to indicate if the original value of the parameter is required. Note that when the action is `SetVariableValue` and when some attributes need to be associated to an action, the localization of the variable or attribute in the code is done only using its name. That is because all the details should have been declared in the measure points section, and the name acts as a reference to that information. We can to reflect on this fact: why do not include the information of the tuning points in its section? The answer is easy: a tuning point is a malleable object, and in addition it could act at the same time as a measure point and as a tuning point. Then, to avoid duplicating the information, we include its declaration only in the measure points section.

4.5.2 Symbols

Once we have discussed the elements needed to define a tunlet, we have to determine the way in which these elements must be declared. In this section we define the symbols which can be used in a valid specification. Then, from the previous analysis, we would need the following categories of symbols:

1. *Delimiters of the tunlet.* All the specification of the tunlet must be declared between the boundaries that determine where the specification tunlet starts and where it finishes.

   - `TUNLET - ENDTUNLET`

2. *Delimiters of sections.* These delimiters are needed to indicate which part of the specification is being defined. As can be seen, these sections do not need `END<SECTION>` delimiter, because on the one hand the inner subsections are delimited and on the other hand the start of the following section indicates the end of the previous one.

   - `MEASURE POINTS`. This section will include variables and values, events, actors, iteration information and performance model parameters subsections.
   - `PERFORMANCE FUNCTIONS`. In this part of the specification, the entities defined are functions to evaluate the deployment of the application.
   - `TUNING POINTS`. The tuning points are the entities that can be defined in this section.

3. *Delimiters of subsections.* All these subsections belong to the measure points section. Even though these delimiters are not strictly needed, they help the user to organize better the specification, due to the measure points section will be in general the biggest section in the specification.

   - `VARIABLES AND VALUES`
   - `EVENTS`
   - `ACTORS`

- `ITERATION INFORMATION`
- `MODEL PARAMETERS`

4. *Delimiters of the different kinds of entities.*

   - `variable-endvariable; event-endevent; actor-endactor; ATTRS`. The three first pairs of delimiters are needed to encapsulate all the information about a particular variable, event or actor and its attributes, which are declared after the key word `ATTRS`.
   - `function - endfunction`. These delimiters are used in the performance functions sections to encapsulate the definition of a particular performance function.
   - `point - endpoint`. Similarly to the previous pairs of delimiters, these are used to define a concrete tuning point.

5. *Labels* for different kinds of information, such as **id**, **type**, **file** or **value**.

   - `id`, will be used to define the name of every entity in the specification: tunlet, events, actors, variables, parameters, points, functions, attributes, etc. The kind of entity which is being named is determined by the delimiters of the entity.
   - `comment`, which is only used for documentation. In case of reuse of a certain tunlet, these comments are useful to "recycle" the specification.
   - `source, type`. The first of these labels is used in *variables* definition to determine what kind of entity in the application is going to be manipulated, for monitoring or tuning. The values this property can take are: 1
     - `asFuncParamValue, asFuncParamPointerValue`, this means the value we are trying to manipulate is part of a function call arguments.
     - `asFuncReturnValue`. In this case, the value is a return value of a determined function.
     - `asVarValue`. This value indicates the entity is a "normal" variable in the application.

- `asConstValue`. Similar to the previous case, this value represents a constant in the application.

The second label describes the basic data type we are managing (`int, short, float, double, char, string`). Structs should be crumbled in its individual fields.

- `actorId`, acts as a reference to the actor's o process' name whose code includes -or has visibility of- the entity (`variable` or `event`) is being defined. The value of this property must correspond to some of the actors' names in the `ACTORS` section.
- `method, class, place, exe`, are properties used to determine the precise place in the source code of the application where the entity we need to manage is located (*variables*, *actors* or *events*). The property `class` is used when the application is written in C++ to indicate what class a method belongs to; when the application has not classes, this property takes the value *none*. The property `place` is used to define where or when an event must be caught, and it can take only one of two values: `entry` or `exit`. In addition it is used when a tuning point encapsulates a `InsertFuncCall` tuning action, where it is necessary specify if the call has to be inserted at the entry or at the exit of the caller function.
- `controliter`. As mentioned in Chapter 2, MATE was conceived to tune iterative applications. Thus, it has to be able to detect when an iteration finishes and the following one starts, with the aim of evaluating the performance model and adapt the behaviour of the application in consequence. However, we have to consider that there are several different machines involved in the execution of the application, where each one is sending the monitoring information to the analyzer as events, but the order in which events arrive can't be ensured. As an instance, we can consider the case in which the event of finalization of the iteration (from the master process) arrives before the event of the finalization of the sending function of some of the tasks in the application (a worker

process); even though the iteration finish had been received, the performance model can't be evaluated until all the needed events arrived from the machines. This is why the events have to include the *controliter* property; it indicates if the event is determining the beginning or the end of a particular iteration, or if it is triggering the evaluation of the performance functions. Note that in the last case, it does not mean that the performance functions will be evaluated: The evaluation will depend on the completion conditions of every actor, due to in general the completion conditions are true when every event has been received and the information for each actor has been completed. The values that this property can take are: {`begin, end, eval, no`}, to indicate beginning or end of an iteration, triggering of model evaluation or indifferent function, respectively.

- `utility`. Due to some instrumentation could be added or removed from the application according to the current conditions of the system, for each event the user has to indicate if the event must be always caught or if it is an addable or removable event. Thus, `utility` can assume three different values: {`always, addable, removable`}. In case the event is `addable`, when required the addition the user has to use the following syntax: `add_event(e)`, where `e` represents the name of the event. Similarly, when an events becomes unnecessary, the user has to specify `rem_event(e)`.
- `min, max` can take *int* values and will be used to determine the size of the internal data structures used to store information related to each particular actor in the application.
- `completion` is a logical expression which will be used to decide if the actor finished an specific iteration or not. This will be used before evaluating the performance functions to ensure every task in the application has finished.
- `inic, value`. These properties are used to define the expressions to calculate the initial value of each attribute and the way in

which they should be updated during each iteration.

- `cum`. This property is used in attributes of iteration information or performance parameters to indicate if the value is defined by some cumulative or comparative operator. This property is taken into account when defining the functionality of CPs and Global Analyzer, in order to implement the logic to preprocess information in CP or reinterpret in Global Analyzer. The supported operations are: cumulative addition or multiplication and minimal or maximal elements. More details are provided in Section 4.5.6
- `dependency` is used to declare the name of the entity the current one is depending on. In general, the evaluation of the value of a particular entity is defined in function of the values of other entities. Thus, the evaluation can only be done once every argument of it had been evaluated. These dependences are used to determine the order of evaluation through the tunlet.
- `depinic` is used similarly to `dependency` but it is used to determine the dependences of the `inic` property.
- `def` includes the definition of the performance function, written in C/C++. Any additional function needed to such definition can be included in the *include* section.
- `syncfunction` is used by the tunlet to indicate the Tuning phase what is the function -the moment in the execution- in which the changes should be introduced. If the changes are independent of the point of execution, this property take the value 0. Note that in case of using synchronization information, the application has to temporally stop the execution to insert a breakpoint, and then restart the execution.
- `syncplace` specifies if the stop has to be made at the entry or at the exit of the function.
- `cond` represents a condition to apply the tuning. Even though the tunlet is provided with the knowledge to decide if the behaviour of the application can be improved -through the performance model-

sometimes there are additional aspects to take into consideration according to the decisions made by the tunlet. For example, when working with the tunlet to tune the number of workers in a Master/Worker application, the total available resources could be a constraint to apply the improvement the tunlet is proposing; i.e., if tunlet propose to use x workers to improve the execution time, but the resources do not cover x it should be controlled to avoid inconsistencies between the information managed by the tunlet and what is happening during run-time.

- `kind` is used to indicate what kind of tuning action encapsulates the tuning point. The possible values are: `SetVariableValue`, `ReplaceFunction`, `InsertFunctionCall`, `RemoveFunctionCall`, `OnTimeFuncCall` and `FuncParamChange`.
- `idx` and `req` are used when the tuning action is `FuncParamChange`, in order to indicate the ordinal index of the parameter in the arguments of the function, and if the original value of such parameter is required, respectively.
- `ATTRS` is used to delimit the list of attributes associated to an actor, an event or a tuning point. The list could be, in the case of events and tuning points, a list of references to some entity declared in the *VARIABLES AND VALUES* section as well as, in the case of actors, a set of attributes where each one is will be used to store information about the actor.

On the other hand, we need define the rules to specify the values of different elements in the specification:

1. *Expressions*, such as the initialization value of an attribute or the body of a performance function. They must be defined as C/C++ expressions, and must be delimited by {`/#, #/`}
2. *Words*, such as the name of a variable or the file of a specific actor. We can express this by means of a regular expression:
   - ([a..z]|[A..Z])([0..9]| [a..z]|[A..Z]|.|_)*

3. *Iterators*, to allow the management of the data related to identical actors, and iterative or cumulative operations over the stored information. Thus, there exists an array of data for each kind of actor in the application. In the array, each element contains the gathered information of a particular task.

4. *Selectors*, to permit determine which part of the information linked to an actor or event we are interested in. (See below).

Iterators and Selectors are used to simplify the task of the user and avoid its involvement in implementation aspects when defining the specification. Then, the expressions used to define the *initialization* (**init**) and *value* (**value**) attributes must be defined by using the user entities included by the user along the specification. Thus, to access each actor's data, we use a positional access, and select the right attribute such as in any data structure by using the dot (*actor[i].attribute*). Information associated with events and iteration information are accessed in a similar way (*event.attribute, iter.attribute*)

4.5.3 Syntax

The syntax of the language determines what the logic order in which information should be sequenced in the specification to ensure a correct interpretation of it is.

```
TUNLET
    name:
    comment:
    include:
MEASURE POINTS
   VARIABLES AND VALUES
   EVENTS
   ACTORS
   ITERATION INFORMATION
   MODEL PARAMETERS
PERFORMANCE FUNCTIONS
TUNING POINTS
ENDTUNLET
```

Figure 4.11: General syntax of specifications

In the previous section, we presented the different symbols needed to describe and delimit each part and entity in the specification. In this section, we will present the way in which these symbols should be combined to obtain a valid specification, i.e. a valid element of the language we are defining. Then, we start defining the order in which sections and subsections should be declared in the specification, as shows Figure 4.11.

```
VARIABLES AND VALUES
        variable
            id:
            comment:
            source:
            type:
            actorId:
        endvariable
```

Figure 4.12: Properties to define a variable or value

```
ACTORS
        actor
            id:
            min:
            max:
            completion:
            class:
            exe:
        ATTRS
            name
            comment:
            type:
            inic:
            depinic:
            value:
            cum:
            dependency:
        endactor
```

```
EVENTS
        event
            id:
            actorId:
            controliter:
            utility:
            method:
            class:
            place:
        ATTRS
            id:
        endevent
```

Figure 4.13: Properties to define actors and events

As can be seen, firstly the measure points must be defined. They include the variables, events, actors, iteration information and performance para-

meters. Then, the performance functions should be defined and finally, the tuning points section must be specified.

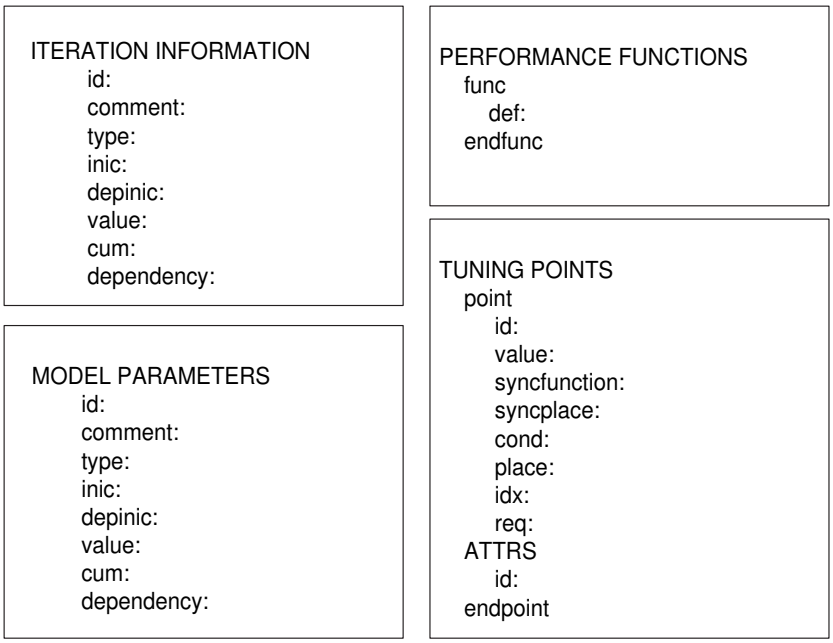

Figure 4.14: Properties to define iteration information, tuning points and performance functions and parameters

Each one of these sections include different kinds of entities described by a set of properties associated to them. Properties which define each entity are listed in Figures 4.12, 4.13 and 4.14.

Note that for each section it is possible declare more than one entity, i.e. for example in subsection *ACTORS* it is possible include several different kinds of actors, where each one of they will be defined by its own set of properties. In order to clarify, see the use cases in Chapter 5.

4.5.4 Grammar

In this section we formally define the Tunlet Specification Language, according to the previous discussion. By following this syntax the user can define the tunlet specifications. In the Section 4.5.5 we explain the semantic associated to each production. Let $\mathcal{G}$ be a grammar which defines the tunlet specification language. Formally, we can define it as the following 4-uple:

$$\mathcal{G} =< \mathcal{N}, \mathcal{T}, \mathcal{S}, \mathcal{P} > \qquad \text{where}$$

1. $\mathcal{N}$ is the set of **nonterminal symbols**.

$$\begin{aligned}\mathcal{N} = \{ \quad & \mathsf{S},\ \mathsf{Tunlet},\ \mathsf{Vv},\ \mathsf{Ev},\ \mathsf{Ac},\ \mathsf{Ii},\ \mathsf{Pp},\ \mathsf{Fc},\ \mathsf{Tp}, \\ & \mathsf{DELIMV},\ \mathsf{VBLE},\ \mathsf{DELIMEV},\ \mathsf{EVENT},\ \mathsf{IDEV}, \\ & \mathsf{DELIMAC},\ \mathsf{ACTOR},\ \mathsf{IDAC},\ \mathsf{DELIMF},\ \mathsf{DELIMP},\ \mathsf{POINT}, \\ & \mathsf{ATTR},\ \mathsf{VBLEREF},\ \mathsf{NUMBER},\ \mathsf{WORD},\ \mathsf{EXP}, \\ & \mathsf{INSTPL},\ \mathsf{TYPE},\ \mathsf{SOURCE},\ \mathsf{CODE},\ \mathsf{UTIL},\ \mathsf{LOG},\ \mathsf{KIND}\ \}\end{aligned}$$

   Some of these symbols observe the following mnemonic rule:

   - $\mathcal{S}$: Specification
   - Vv: Variables and Values
   - Ev: Events
   - Ac: Actors
   - Ii: Iteration Information
   - Pp: Performance model Parameters
   - Fc: Performance Function
   - Tp: Tuning points
   - EXP: represents a C/C++ function or expression, which we consider pre-established.

2. $\mathcal{T}$ is the set of **terminal symbols**, disjoint from $\mathcal{N}$.

$\mathcal{T}$ ={ TUNLET, ENDTUNLET
MEASURE POINTS, PERFORMANCE FUNCTION,
TUNING POINTS, VARIABLES AND VALUES,
EVENTS, ACTORS, ITERATION INFORMATION,
MODEL PARAMETERS, ATTRS,
variable, endvariable, event, endevent,
actor, endactor, func, endfunc, point, endpoint,
id, min, max, completion, exe,
class, method, place, comment, source,
actorId, controliter, utility, def,
kind, syncfunction, syncplace, cond, idx, req,
begin, end, eval, no, none, include,
always, addable, removable,
type, inic, value, cum, entry, exit, dependency,
depinic, int, short, float, double,
char, string, true, false,
asFuncParamValue, asVarValue, asFuncReturnValue,
asConstValue, asFuncParamPointerValue,
SetVariableValue, ReplaceFunction,
InsertFunctionCall, RemoveFuncCall,
OnTimeFuncCall, FuncParamChange,
[0..9],[a..z], [A..Z], _, ., (,), :, /, [,], ;, * }

3. $\mathcal{P}$ is the set of **production rules** in the language.

$\mathcal{P}$: {

$\mathcal{S} \longrightarrow$ TUNLET Tunlet
MEASURE POINTS Vv Ac Ii Pp
PERFORMANCE FUNCTIONS Fc
TUNING POINTS Tp
ENDTUNLET

Tunlet	$\longrightarrow$	`id:` WORD
		`comment:` $(\text{WORD})^+$
		`include:` $(\text{WORD})^+$
Vv	$\longrightarrow$	$(\text{DELIMV})^+ \mid (\texttt{include}\ \text{WORD})^+$
Ev	$\longrightarrow$	`EVENTS`$(\text{DELIMEV})^+$
Ac	$\longrightarrow$	`ACTORS` $(\text{DELIMAC})^+$
Ii	$\longrightarrow$	`ITERATION INFORMATION` $(\text{ATTR})^+$
Pp	$\longrightarrow$	`MODEL PARAMETERS` $(\text{ATTR})^+$
Fc	$\longrightarrow$	$(\text{DELIMF})^+$
Tp	$\longrightarrow$	$(\text{DELIMP})^+$
DELIMV	$\longrightarrow$	`variable` VBLE `endvariable`
VBLE	$\longrightarrow$	`id:` WORD
		`comment:` $(\text{WORD})^+$
		`source:` SOURCE
		`type:` TYPE
		`actorId:` WORD
DELIMEV	$\longrightarrow$	`event` EVENT `endevent`
EVENT	$\longrightarrow$	IDEV $(\texttt{ATTRS}\ (\text{VBLEREF})^+)?$
IDEV	$\longrightarrow$	`id:` WORD
		`actorId:` WORD
		`controliter:` CODE
		`utility:` UTIL
		`method:` WORD
		`class:` WORD
		`place:` INSTPL
DELIMAC	$\longrightarrow$	`actor` ACTOR `endactor`
ACTOR	$\longrightarrow$	IDAC $(\texttt{ATTRS}\ (\text{ATTR})^+)?$
IDAC	$\longrightarrow$	`id:` WORD
		`min:` NUMBER
		`max:` NUMBER
		`completion:` EXP
		`class:` WORD
		`exe:` WORD+

DELIMF ⟶ `func def:` EXP `endfunc`

DELIMP ⟶ `point`
VBLEREF POINT (`ATTRS` (VBLEREF)$^+$)?
`endpoint`

POINT ⟶ `value:` EXP
`kind:` KIND
`syncfunction:` WORD
`syncplace:` INSTPL
`cond:` EXP
(`idx:` NUMBER
`req:` LOG) ?
(`place:` INSTPL) ?

ATTR ⟶ `id:` WORD
`comment:` (WORD)$^+$
`type:` TYPE
`inic:` EXP
`depinic:` WORD | `none`
`value:` EXP
`cum:` LOG
`dependency:` WORD | `none`

VBLEREF ⟶ `id:`WORD

NUMBER ⟶ [`0..9`]$^+$

WORD ⟶ ([`a..z`]||[`A..Z`])([`0..9`]||[`a..z`]||[`A..Z`]||
. | _ | (|) | : | / | * | [|] | ;)*

EXP ⟶ C/C++FUNCTION |C/C++EXP

INSTPL ⟶ `entry` | `exit`

TYPE ⟶ `int` | `short` | `float`
| `double` | `char` | `string`

SOURCE ⟶ `asFuncParamValue` | `asVarValue`
| `asFuncReturnValue`| `asConstValue`
| `asFuncParamPointerValue`

CODE ⟶ `begin` | `end` | `eval`| `no`

UTIL ⟶ `always` | `addable` | `removable`

$$
\begin{aligned}
\text{LOG} &\longrightarrow \texttt{true} \mid \texttt{false} \\
\text{KIND} &\longrightarrow \texttt{SetVariableValue} \mid \texttt{ReplaceFunction} \\
&\quad \mid \texttt{InsertFunctionCall} \mid \texttt{RemoveFuncCall} \\
&\quad \mid \texttt{OnTimeFuncCall} \mid \texttt{FuncParamChange}
\end{aligned}
$$

}

4. $\mathcal{S}$ is the **start symbol** of the gramar.

4.5.5 Semantics

In order to define the semantic of the Tunlet Specification Language, we use a **syntax-directed definition**. According to [1], a syntax-directed definition is a generalization of a context-free grammar in which each symbol has a set of associated attributes. If we consider each node in a syntax tree represents a grammar symbol through a register with fields to store information, then each attribute of the grammar symbol corresponds to the name of one field. The value of an attribute is defined by a semantic rule associated to the production. Formally:

> In a syntax-directed definition, each production $A \longrightarrow \alpha$ has associated a set of semantic rules $b := f(c_1, c_2, ..., c_k)$, where:
>
> - f is a function, **AND**
> - b is a synthesized attribute of A and $c_1, c_2, ..., c_k$ are attributes belonging to the symbols of the production, **OR**
> - b is a inherited attribute of some symbol in α and $c_1, c_2, ..., c_k$ are attributes belonging to the symbols of the production.

In other words, we will augment the grammar presented in Section 4.5.4 with some attributes and semantic rules to explain the semantics of the language. In particular, we used a *synthesized attributes definition*, i.e. we will use only synthesized attributes.

In the following, we present the syntax-directed definition, and then we explain and define -in pseudocode- the procedures used in the translation process.

Syntax Directed Definition

Even though we provide the whole syntax-directed definition, the most interesting semantic rule is associated with the start symbol $\mathcal{S}$. This semantic rule condenses the operation of the translation from the specification to the tunlet. The remaining rules, in general, allow for obtaining -*synthesizing*- the values of the attributes.

For legibility and space reasons, in this section we just present the starting production with its associated semantic rules. The whole syntax directed definition is documented in Appendix A. Note that for space reasons too, we present the semantic rules following the right part of each production, rather than in a table as proposed in [1].

$\mathcal{S} \longrightarrow$ `MEASURE POINTS` Vv Ac Ii Pp
`PERFORMANCE FUNCTIONS` Fc
`TUNING POINTS` Tp
{
Translate_and_Solve_Dependences()
Create_Tunlet_Stats()
Create_Tunlet()
}

In the next paragraphs we describe the semantic of each procedure involved in the semantic rule.

Procedures

Before starting with the definition of the semantic, some assumptions have to be established. In the semantic rules, we assume that every symbol which manages a list of some "x" entity, (attribute called *xs*) has an attribute "i", initialized in 0, used to indicate the position in the list where the next "x" should be stored.

Translate_and_Solve_Dependences: this is a procedure used to translate the entities used by the user through the specification into entities

of MATE. In addition, the dependences in the evaluation order of attributes and parameters are determined. For legibility reasons, in the algorithm we abbreviated as *attr.value* the reference to the corresponding synthesized attribute of *attr* which should be:

Ac.actors[j].attrs[Ac.actors[j].k].value

where j selects the j^{th} actor from the list of actors and k selects the k^{th} attribute from the list of attributes. Similarly, we represented attributes in `Ii`, parameters in `Pp`, tuning points in `Tp`, variables in `Vv` and events in `Ev`, by using *attr.value*, *param.value*, *point.value*, *vble.type* and *event.sent*, respectively.

```
Translate_and_Solve_Dependences()
{
  for each actor in Ac.actors
    for each attribute in the actor
      Translate(attr.value)
      ObtainDependences()
    Translate(actor.comp_cond)
  for each attribute in Ii.attrs
    Translate(attr.value)
    ObtainDependences()
  for each parameter in Pp.attrs
    Translate(param.value)
    ObtainDependences()
  for each tuning point in Tp.points
    Translate(point.value)
  for each variable in Vv.vbles
    Translate(vble.type)
  for each event in Ev.events
    event.sent=ObtainSentences()
  for each function in Fc.funcs
    Unfold()
}
```

As mentioned in Section 4.5.2, to simplify the task of the user and avoid

its involvement in implementation aspects, the expressions used to define the *initialization* (**init**) and *value* (**value**) attributes must be defined by using the user entities included by the user along the specification. Thus, to access each actor's data, we use a positional access, and selecting the right attribute such as in any data structure by using the dot (*actor[i].attribute*). Information associated with events and iteration information are accessed in a similar way (*event.attribute, iter.attribute*). The *Translate()* action is used to translate a string containing user-entities to an equivalent string including MATE entities. The main transformations are presented in figure 4.15 as a translation scheme. "e" represents the name of a scanned event, "attr" represents a given attribute of an actor, event or iteration, "a" represents the name of an actor or the string "iter".

Token-String	Translation
e.	print(r.)

attr	if (timestamp) print("GetTimeStamp()")
	elseif (task) print("GetTask()")
	elseif (id) print("GetId")
	else
	find position of attr in event
	print("GetAttributeValue(pos)")

a[	print("aData d =") Obtain_Position()
].attr	print("d.Set_attr(") Obtain_argv()

Figure 4.15: Translation Scheme for transforming user entities in MATE entities

These translations are determined and dependent on the implementation of MATE. Suppose the user refers to some attribute of an event as "e.attr". In MATE, the attributes of an event can be only managed when the event is received and it is handled by the event handler, which in general is the tunlet. Then, in the *HandleEvent* method, *r* represents the event which provides a series of methods in order to obtain the information associated to it (see Chapter 2). Given that each event has three default attributes *timestamp*, *id* and *task* indicating the instant, the process identifier and the task in which it happened, "attr" is translated using the default me-

thods *GetTimeStamp*, *GetId* or *GetTask* when *attr==timestamp*, *attr==id* or *attr==task*, respectively. When the attribute is not a default one, the *GetAttributeValue* method is used to obtain the attribute from the event. The position "*pos*" is inferred from the order followed by the user in defining the attributes associated to the event. A string such as "a[i].attr" is translated in several steps. In first place, the corresponding data structure in the tunlet must be obtained to manage the information. In the implementation of MATE the data structure used to store the information associated to actor *a* is called "*aData*". In a following step, the index of such structure is obtained in order to manage the corresponding structure. In other step, the methods associated to *aData* are used in order to obtain the value of *attr* (when the string is a r-value) or in order to set the value (when the string is a l-value), such in the table where *d.Set_ attr* is used; in the following step, the argument for such setting method is obtained. The particular translation of tuning points interprets the value of each property according to the *kind* of associated tuning action.

Obtain_ Dependences() is a function used to solve the chain of dependences among the different entities in the specification. In general, the chain of dependences starts depending on an event, and is used to determine the order of evaluation of the attributes and performance parameters. The *dependency* and *depinic* properties are involved in this process.

Obtain_ Sentences() is a function used to associate the corresponding set of actions to execute when an event is handled. As mentioned above, the tunlet is the event handler, and in consequence it is responsible for extracting, processing and classifying the information carried by the event, to constitute the iteration information, actors' attributes and then the performance parameters. The set of actions is established according to the dependences determined in the previous procedure. The symbol "||" represents the concatenation of strings.

```
Obtain_Sentences()
{
  according to Obtain_Dependences()
  for each attribute depending on the event
```

```
    event.sent:=event.sent||attr.value
}
```

Unfold is a function used to obtain each part of the performance functions definition: the returning value, the name, the parameters, the definition. They are used to define the corresponding methods in the tunlet.

Create_Tunlet_Stats: is used to generate the statistics structures used to store the information collected during run-time. The information can be related to each actor or to each iteration. The *Set* and *Get* methods are defined to assign or obtain the value of a particular attribute of the class. Such classes are used by the tunlet to store and obtain the values.

```
Create_Tunlet_Stats()
{
  //Create individual structures for each actor
  for each actor in Ac.actors
    create a class to manage its attributes
    for each attribute in Ac.actors[j].attrs
      declare the attribute as a class-attribute
      create Set and Get methods
    create an isCompleted() method
      using Ac.actor[j].comp

  //Create individual structures for each iteration
  create a class to manage the attributes of iterations
  for each attribute in Ii.attrs
    declare the attribute as a class-attribute
    create Set and Get methods
  for each actor in Ac.actors
   create a map <int,actor_class>
   create Set and Get methods
}
```

Create_Tunlet: in this procedure, the entities in the specification are

used to constitute the parts of the tunlet which implement the logic to obtain and evaluate the elements in the performance model.

```
Create_Tunlet()
{
  for each actor in Ac.actor
    Create_Instrumentation_Methods()

  for each parameter in Pp.attrs
    declare as a tunlet class attribute
    Create_Updating_Methods()

  for each function in Fp.funcs
  Create_PerfFunc_Methods()

  for each tuning point in Tp.points
  Create_Tune_tp_Methods()

  Define_HandleEvents()
}
```

When a specific task starts execution, the tunlet has to require the corresponding instrumentation to Application Controller (more specifically to Monitor).

```
Create_Instrumentation_Methods()
{
  for each actor in Ac.actors
    for (each event in Ev.events such that
           Ev.events[k].actor==Ac.actors[j])
      create the requirement:
         include the attributes in Ev.events[k].vblesr
      send the requirement to Application Controler
}
```

When an iteration finishes, the performance parameters can be updated to evaluate the performance model. Thus, the tunlet has to include some methods to update the values. Similarly, the performance functions have to be implemented as methods of the tunlet.

```
Create_Updating_Methods()
{
  for each parameter in Pp.attrs
    use Pp.attrs[h].value to define the body
                                of the method
}

Create_PerfFunc_Methods()
{
  for each parameter in Fp.funcs
    use Fp.funcs[m].name, Fp.funcs[m].type,
        Fp.funcs[m].param and Fp.funcs[m].def
        to define the method
}
```

In the loop of reception of events, the Global Analyzer and the CPs have to include the logic to use the information carried by the event in order to assign the corresponding values to the entities depending on such an event.

```
Define_HandleEvents()
{
  for each event in Ev.events
    use Ev.events[n].sent to define the
      corresponding management of the event
}
```

An special aspect in the semantic interpretation of the tunlet specification is the inference of the logic to the CPs preprocess the information associated to the events and the logic to the Global Analyzer to interpret the information received from each CP. We will analyze such logic in the following section.

4.5.6 Deducing CPs logic from the specification

In Chapter 3 we discussed the needs for providing scalable qualities to the Analyzer, and what will be the strategy to reach such a configuration. In this section, we describe the mechanisms used to transform the information included in the specification of the tunlet in a useful way to automate the creation of the different processes involved in the analysis phase: *Collector-Preprocessor* and *Global Analyzer*.

Before to expand on the obtaining of the code from the specification, we should have in mind the elements of the specification, summarized in Figure 4.11. The *MEASURE POINTS* section condenses all the information needed to evaluate the deployment of the application. Just to remind and summarize the main parts, it includes variables, performance parameters, iteration information, actors and events. The last two categories can have associated a set of attributes. Performance parameters, iteration information and attributes are defined in a similar way, then in order to make easier the discussion, we will call all of them as "*attributes*".

Each one of the attributes has a property named `value` which indicates the way in which the value of the entity should be calculated. In general, that value depends on other attributes or variables in the specification. Unfortunately, the attributes the value depends on are not always associated to the same actor the attribute is, and this fact could complicate the mechanism to calculate it. The trouble is mainly in the fact that we need to divide the collecting and preprocessing of the information incoming from the tasks in several different groups, and this division could not match with the original conception of the performance model as a whole. If some information belonging to another group of tasks is needed, in order to obtain it we must to pay for the cost of communication. Then, the idea is avoid this hardship by calculating what locally is possible and to postponing the global evaluations comprising the different sets of tasks information to the Global Analysis phase.

CPs and Global Analyzer must include data structures and mechanisms to support and process local data and to prepare data needed to the global evaluation of the model. Then, the collection of data will be done by using

the same data structure as in centralized approach. As shows Figure 4.16 such data structure includes a field to store each one of the attributes associated to the entity. This particular example constitutes the data structure to store the master attributes, such as `firstSend, lastRecv` and `lastWorker`. In the case of actors whose property `max` is greater than one, the data structures are managed as an array, where each element contains the information of a particular actor. Iteration information and performance parameters have their own data structure.

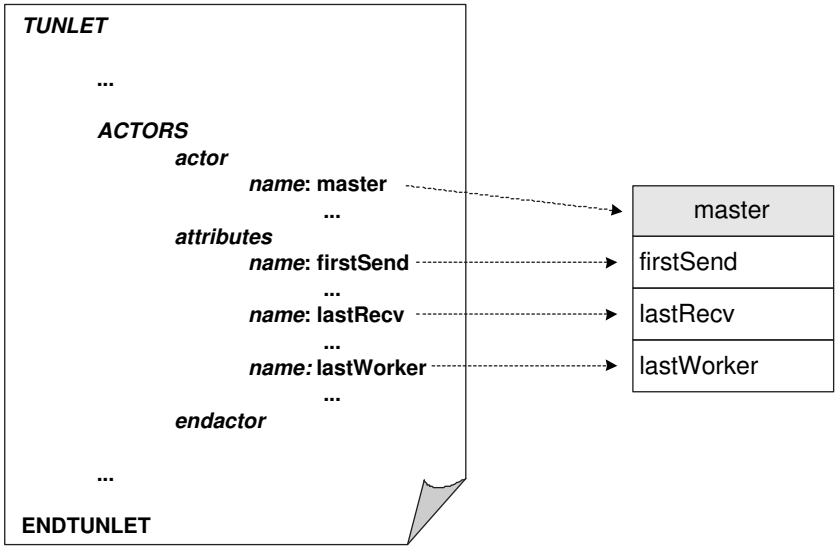

Figure 4.16: Data structure to store the attributes of a particular actor

If we study in depth the situation of the `lastWorker` attribute, the master process needs to know which of the workers was the last in send back to him the results of the calculations. In the specification it is defined as in Figure 4.17. As can be seen, `lastWorker` attribute depends on `WRepliesM` event. Let suppose that this event is caught in the worker code, when a send action is executed. Obviously, in Analyzer's centralized approach to obtain this information is relatively easy because the only Analyzer is receiving all the events. But in the new approach, workers' events are collected by CPs. Then, each CP is capable of determining which of the workers the last in sending the answer to it was. That will constitute a "local `lastWorker`", but the master will be who decide which of those partial `lastWorkers` is the

global `lastWorker`.

```
...
name:lastWorker
type: int
inic: lastWorker=0
value: if(WRepliesM.timestamp>master.lastRecv || master.lastRecv==0.0)
                    master.lastWorker= WRepliesM.id;
dependency: WRepliesM
...
```

Figure 4.17: Specification of lastWorker attribute of master actor

Here arises the need of including a new kind of structure to manage the data needed by other entities. Then, in addition to the array of data structures, we need a data structure called `Aux` with a field to store each one of the values needed by foreign entities. On the one hand, the way in which that value will be initialized and calculated in the CPs will depend on the definition of the entity. In our example, `Aux` will provide a `lastWorker` field which is initialized in the same way as `master.lastWorker`. Then, due to the calculation of the `value` of this attribute is only involving local information (i.e. workers information) can be redefined in the same way as the global `lastWorker`, as presents Figure 4.18.

```
Aux
  field_name:lastWorker
  field_type: int
  field_inic: lastWorker=0
  field_value: if(WRepliesM.timestamp>Aux[lastRecv] || Aux[lastRecv]==0.0)
                    Aux[lastWorker]= WRepliesM.id;
  field_dependency: WRepliesM
  ...
```

Figure 4.18: Aux data structure in subordinated Analyzers

On the other hand the global Analyzer have to redefine the determining of the `lastWorker`. To do that, it will need an equivalent Aux structure to store the data sent by each CP, then it will manage an array of such structures. A mechanism to compare `lastWorker` fields will be needed in

order to determine the global worker who replied at last.

In the translation from a specification into a tunlet, the logic to constitute the collecting, preprocessing and reinterpretation of the information sent by the CPs to the Global Analyzer is inferred from the declarations in the specification, specially by considering the value of the property *cum*. As introduced in Section 4.5.2, this property should be *True* when the value of the attribute under consideration is calculated using some cumulative operation (addition or multiplication) or comparative operators (minimal or maximal elements). However, even though the *cum* property is *True*, the translator decides if the attribute can be effectively preprocessed by studying what actors the value depends on. In other words, the value of an attribute *att* could depend, for example, on a cumulative operation over a kind of actors (the cumulative addition of attribute *att0* of every worker) and a particular attribute of another kind of actor (the division by the attribute *att1* of the master), as follows:

```
id:att
type:int
inic:att=0
depinic:none
value: int i;
       for(i=0;i<n;i++)
       {att+=worker[i].att0;}
       att=att/master[0].att1;
cum:true
dependency: ...
```

where n represents the total amount of workers. If the information -the events- related to each kind of actors is not collected by the same CP, the value of *att* cannot be calculated at CP level even though *cumm*==*true*. This force the CP to send all the information to the Global Analyzer, increasing the Global Analyzer processing time.

A technique to solve these limitations is to provide some auxiliary attributes to calculate the cumulative/comparative operations and then use

the result in the original *att*, as in the following example, where n will be interpreted by the CP as the number of workers locally managed:

```
id:attaux
type:int
inic:att=0
depinic:none
value: int i;
       for(i=0;i<n;i++)
       {attaux+=worker[i].att0;}
cum:true
dependency: ...


id:att
type:int
inic:att=0
depinic:none
value: att=attaux/master[0].att1;
cum:false
dependency: attaux
```

In this case, *attaux* can be partially obtained at each CP and the final treatment of *att* will be done at Global Analyzer level. Recapping on the *Aux* structure, it should include an attribute *"attaux"* in which the CP will store the correspondig value, and which will be reinterpreted by Global Analyzer to complete the information, before evaluating the performance model.

Note that given the complexity that could be involved in the definition of the *value* property, we require the cooperation of the user (by including auxiliary attributes as before) in order to maximize the amount of information preprocessed in CPs.

4.6 Tunlet Generator Implementation

Once the tunlet has been specified, it is possible translate such specification in source code. Such as in all translations of code, we follow a series of steps to obtain the source code of the tunlet, ready to be incorporated in MATE.

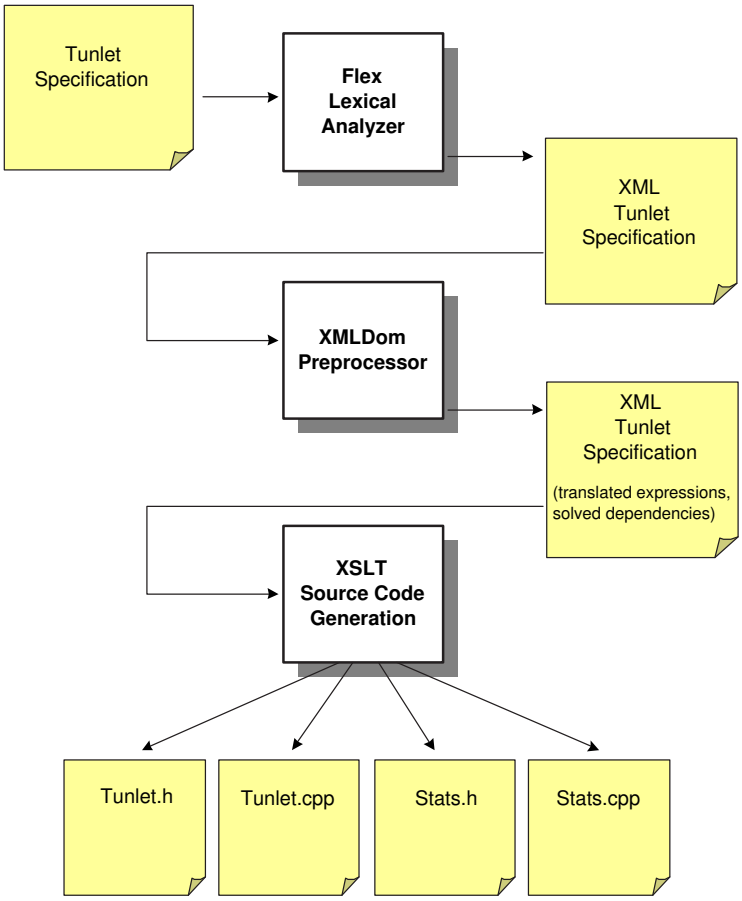

Figure 4.19: Phases in the generation of a tunlet from a specification

The successive phases are illustrated in Figure 4.19 and are described in the following paragraphs:

1. *Lexical Analysis*: The input of the analyzer is the specification of the tunlet, written in a text file. The output is an equivalent specification but following the XML syntax [61].

   This step exists for user-friendliness reasons, and consists in translating the specification into its equivalent XML specification. The lexical

analyzer was implemented in Flex [62]. By considering the semantic defined in the previous section, the lexical analyzer obtains the initial value of the attributes.

2. *Preprocess*: the input is the XML specification of the tunlet (obtained in the previous lexical analysis phase). In this phase, existing dependences among attributes and events through the specification are solved (i.e. ordered to avoid inconsistencies in the behaviour of the tunlet); in addition, the expressions to calculate the initialization and value of each entity are translated into internal structures of MATE. The output is the same XML specification but with the expressions translated and the dependences solved. The preprocessor is an XMLDom program [58]. By considering the procedures presented in the previous section, the preprocessor implements the *Translate_ and_ Solve_ Dependences* procedure.

3. *Source Code Generation*: the input is the XML specification obtained in preprocess phase. The output is a set of C/C++ files, with the source code of the tunlet. This last step in generating a tunlet, consists in extracting information from the different sections of the specification to conform the source code of the tunlet. The generator was implemented as several XSLT stylesheets [72, 71]. The *Create_ Tunlet_ Stats* and *Create_ Tunlet* procedures are implemented in this step.

Notice that the generation process includes several steps, but the user only is involved in the definition of the specification. From that specification it is possible generate the source code.

4.7 What does the User need to know?

The aim of this section is to summarize what explained previously, and to condense the main concepts needed to understand and use MATE, the methodology to develop tunlets, the Tunlet Specification Language and the Tunlet Generator, taking into account the view of the user.

First at all, recapping on Chapter 2, **MATE** *(Monitoring, Analysis and Tuning Environment)* is, as its name indicates, a tool conceived to control

and adapt the execution of parallel iterative applications. This environment works in an automatic and dynamic way, characteristics especially useful in both situations: when the user is not an expert on performance analysis and/or when the applications are executed in heterogeneous or time-sharing environments or their behaviour depend on the input data. The general functioning of MATE is shown in figure 4.20.

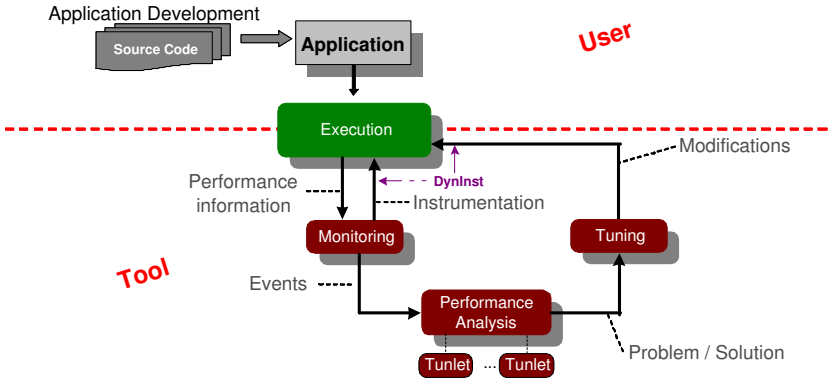

Figure 4.20: General operation of MATE

In this process, the user is only involved in the development of the application. In general, MATE can solve every problem which can be expressed by means of a performance model. Performance models constitute the knowledge used by MATE to conduct what information is needed to collect during the execution (so called *measure points*), how to evaluate the collected information (the *performance functions*) and which are the changes needed to tune the application (the *tuning points*). Each performance model is encapsulated in a piece of software so called "tunlet". As shows the figure, the tunlets are part of the analysis phase, and are in fact which own the knowledge.

To use MATE, the user can be in one of the following situations:

- His application presents a particular performance problem which has not been included in MATE yet.
- His application has some of the performance problem whose solution is included in MATE but the corresponding tunlet was implemented according to another application.

In both previous situations, the user can use the methodology depicted in Section 4.3 in order to identify the abstractions defined in Section 4.2 needed to generate a tunlet. The **Tunlet Specification Language** can be used in order to specify a new performance problem to be transformed in a tunlet, or in order to reinterpret an existing specification for the user's application. When the specification is ready, the **Tunlet Generator** automatically generates the tunlet from the specification. In the following we present the algorithm to add a tunlet to MATE. In the next subsections, we explain each step in more depht.

1. To Develop the parallel application
    - Following the methodology chosen by the user
    - Using the Master/Worker framework included in MATE, which already provides the measure points for the tunlet.
2. To define the tunlet abstractions
    - Following the methodology 4.3
3. To write the specification of the tunlet
    (a) Defining the measure points
        - Defining the variables
        - Defining the events
        - Defining the actors
        - Defining the iteration information
        - Defining the performance parameters

    (b) Defining the performance functions

    (c) Defining the tuning points
4. To generate the tunlet
    - Executing the TG script
5. Execute the application under the control of MATE in the cluster

4.7.1 Developing the parallel application

MATE can be used to tune any parallel and ***iterative*** application. Currently, applications have to be ***written in C/C++*** and have to be based on ***pvm*** message passing library. The applications can be developed following any methodology or using any development tool, whenever they respect those three requirements to be tuned by MATE. In particular, as mentioned in the algorithm, MATE includes a framework to the semi-automatic development of Master/Worker applications. Then, if the user develops his Master/Worker applications by using this framework, not only he straightforward can use the pool of tunlets predefined, but he also is exempted from think about some aspects of the specification of the tunlet when defining a new one, such as the measure points, due to they are provided beforehand.

4.7.2 Defining the tunlet abstractions

Once the application had been developed, the user should follow the proposed methodology to understand the application and the performance model in the terms that MATE does. Once the user identified the abstractions according to Section 4.2, they can be formalized in a specification to its later translation into a tunlet.

4.7.3 Writing the specification of the tunlet

In order to write the specification of a tunlet according to some given application and performance model, the user can follow the pattern presented in Figure 4.21 at the end of this section (4.7). In the following we provide some further details about different parts of the specification.

1. Defining the **variables and values**, including the identifier, data type, the name of the actor which has visibility of it and if it is a variable, parameter, function result, etc.

2. Defining the **actors** involved in the application. For each actor, it is needed to declare the name, the class in which is included or defined, and the name of the executable file. Some additional information is required:

- the minimal and maximal quantity of this type of actor could co-execute during the execution of the application. These numbers are needed to generate the structures to manage the behavioural information of each process along the successive iterations of the application.
- a completion condition to detect when the actor reached the end of its tasks along an iteration.
- the actor's attributes, i.e. the properties that should be registered in each iteration.

3. Defining the **events** to capture. In general, each event is defined by its name, the actor it is associated with (i.e. where it happens) and the place in the source code. By considering that MATE is based on iterations and the events constitute the means to send information about the application to the Analysis phase, it is through some events that the beginning and end of the iterations are captured. In addition, it is needed to consider that the evaluation of the performance functions is not only depending on the reception of the event indicating the end of the iteration, but it depends also on the reception of every data needed to evaluate the functions. Then, due to the order in which events arrive from different machines can't be ensured, the user has to indicate what the events which are finishing the iteration for each actor are. In this way, when one of these events arrive to the Analysis phase, a query is made in order to verify if the information is complete or if it is needed to wait for additional events. Thus, the events controlling the beginning or the end of the global iteration, and the end of an interation for each actor, have to be indicated with value `begin`, `end` or `eval` respectively in the property *controliter*. When the event has not any special consequence, the value of *controliter* is `no`.

   A particular event can carry associated some specific information in its attributes. Note that the attributes associated to a particular event have to be visible from the actor's code where the event is caught. The quantity of bytes sent, can be an interesting metric caught when an event that indicates the exit of the sending function occurs. Each event

has three default attributes: *timestamp*, *tid* and *task* to indicate the specific instant, the process id and the task it was caught in.

4. Defining the **performance parameters** of the performance model, whose value generally are calculated as a function of the attributes of actors or events.

5. Defining the **iteration information** necessary to describe the behaviour of each iteration, not included in the other sections, such as the total communication or processing time in the iteration. This section requires the definition of an attribute to indicate the current iteration.

In general, all the elements in the specification with a set of attributes, must declare for each attribute its name, the data type, the initialization value and the way to update its value (see below). Finally, if the attribute depends on another attribute or event -in the sense that the value of the current entity can't be calculated until the one it depends on was determined- the name of such entity should be expressed.

6 Defining the **performance functions** as the implementation in C/C++ language of the performance expressions of the model. As in every element through the specification, the functions will depend on entities included in the specification. The necessary mathematical libraries have to be declared in the beginning of the specification, in *include* section.

7 Defining the **tuning points** declaring the kind of action, the name (id) of the involved variable or function, the new value to be assigned, a condition to apply the tuning, and some information about synchronization: the appropriate place and instant to change the value of the point. In case the kind of tuning action consist in changing the value of a function parameter (`FuncParamChange`) two additional properties are necessary: the ordinal position of the parameter in the list of arguments of the function (the *idx* property) and a boolean value to indicate if the original value of such parameter is needed (the *req* property). In case the tuning action is `RemoveFuncCall` the *value* property indicates the

name of the caller function where the function call have to be removed. In case the tuning action is `InsertFunctionCall` the *value* property indicates the name of the function or method where the call have to be inserted. In this case, the additional property *place* (entry or exit) have to be specified. In addition the attributes of the function have to be expressed as well as when the tuning action is `InsertFunctionCall` or `OnTimeFuncCall`.

To simplify the task of the users and reduce their involvement in implementation aspects, the expressions used to define the *initialization* (**init**) and *value* (**value**) properties (for *attributes*, *iteration information* and *performance model parameters* sections) must be defined by using the user entities (variables, events, actors, etc.) included along the specification. Thus, to access each actor's data, we use a positional access, and select the right attribute such as in any data structure by using the dot (*actor[i].attribute*). Information associated with events and iteration information are accessed in a similar way (*event.attribute, iter.attribute*).

4.7.4 Special considerations and constraints

When defining the abstractions and the specification, the user should consider some special requirements of MATE and/or some constraints of the language:

- The start and the end of each iteration must be detected. Therefore, the user has to provide the definition of two events to capture such instants. Each one has to indicate its functionality in the *controliter* property, indicating if the event detects the `begin` or the `end` points of the iteration. These events are necessary to detect when the performance model can be evaluated.

- Each event should indicate what is the iteration in which it was caught. In general, this information can be obtained from some variable in the application which can be associated to the event as an attribute. This information is used to classify the information carried by the event in the corresponding iteration data structure.

- Even though the user have to define the dependences among different entities (using the *dependency* and *depinic* properties), in addition we recomend to declare the entities following the dependence order whenever it is possible. Note that the order of the specification sections should not be changed, the dependence order have to be considered to entities in the same section.

- Some special functions can be used to program some functionalities when defining the *value* property:

  - `add_event(e)` and `rem_event(e)`, where `e` represents the identifier of an event. These functions can be used when the insertion of an additional event is required (whenever the *utility* property of the event is `addable`) or when the removal of some event is necessary (whenever the *utility* property of the event is `removable`).

  - `Tune_<tuningpoint_id>()`. In general, this family of functions is not necessary when the default cycle of collection of data, analysis and tuning actions is enough in order to cover the necessities of the tunlet. However, some performance models require multiple tuning actions along an iteration, applied under certain conditions. Thus, these functions can be used when a tuning action have to be applied in some instant different from the normal and automatic one.

- When some point in the code has to be specified, the `method`, `class` and `place` properties are used, in the case of specifying an event. However, if the implementation of the application is not based on the object oriented paradigm, the user has to just provide the name of the corresponding function in the `method` property, and `class` remains empty. In the case of tuning points, most of the available tuning actions require the name of the involved functions. When the application is based on the object oriented paradigm and the function under consideration is a method, the user has to provide the name as the combination of *<class>::<method>*.

- In order to take a major advantage of the preprocessing (in CPs) of

the information associated to the events, we recommend considering the use of auxiliary actors' attributes, iteration information and performance parameters when the `cum` property of an entity is `true` but the `value` depends on entities belonging to different kinds of actors. In other words, the user should try to separate the calculus of cumulative additions or multiplications in an independent entity, and then use the result in the original entity. The same has to be considered for comparative relations.

4.7.5 Generating the tunlet

The command used to automatically generate the tunlet from the specification is the following:

```
> TG <specification_path>
```

TG is a script which supervises the successive stages of the generation of the tunlet. Those phases are *lexical analysis*, *preprocessing code generation* and *compiling*, but they are not important from the user's point of view.

4.7.6 Executing the application under MATE

The general way in which the application has to be executed under MATE is by using the following command:

```
> MATE <application_path> <application_arguments>
```

Pvm should be started before the previous command. Depending on the particular functioning or manager of the execution system (the cluster in which the application will be executed and dynamically tuned) the way in which the user prepares the environment can be manual or automatic. Thus, the user should consult the administrators of the systems about the use of MATE. Another consideration is that MATE requires some additional machines which will be used exclusively for the analysis phase; then, when asking for resources to the system (number of hosts in pvm) it is needed to include these additional machines.

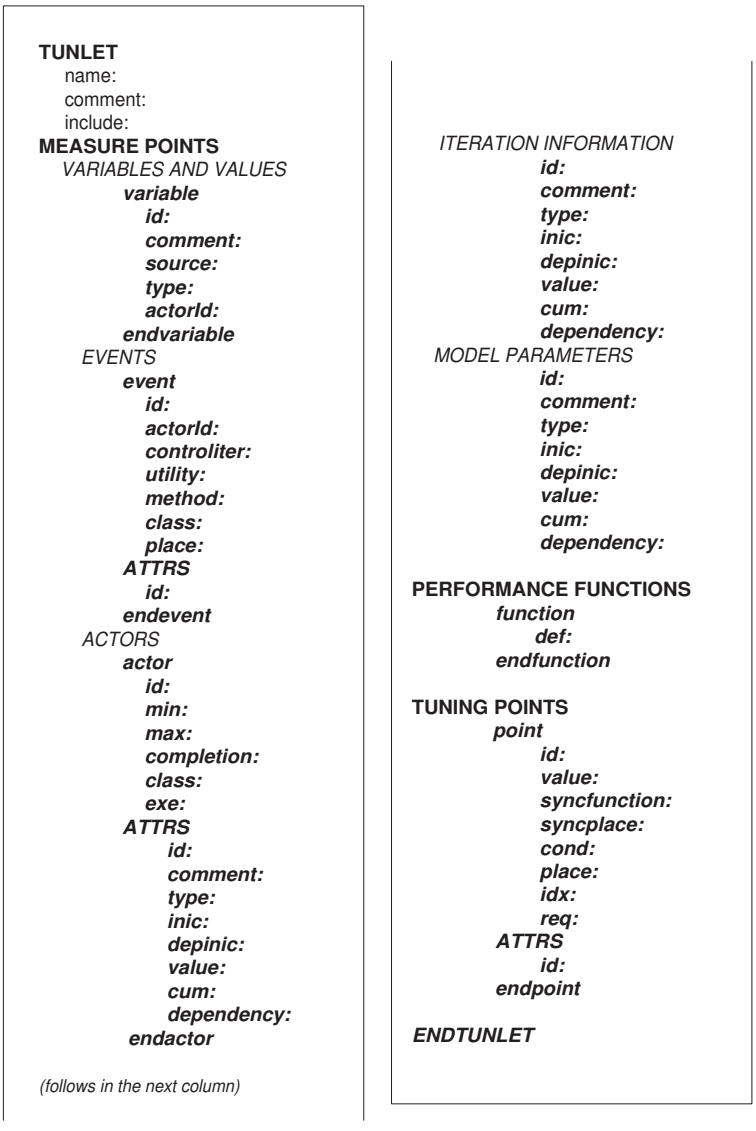

Figure 4.21: Template to specify a tunlet

Chapter 5

Use Cases

> "[...] los Espila se avinieron a iniciar los experimentos, y Elena se dedicó muy en serio a estudiar galvanoplastia mientras que el sordo preparaba los baños y se ponía práctico en ese trabajo de unir en serie o tensión los cables del amperímetro y en manejar la resistencia. Hasta la anciana participó en los experimentos y nadie dudó, cuando consiguieron cobrear una chapa de estaño, que en breve tiempo se enriquecerían si la rosa de cobre no fracasaba."
>
> *Los Siete Locos*, Roberto Arlt

IN this chapter we present two examples on how to specify tunlets from some given performance models. The models we present are defined for the Master/Worker programming scheme and are the following:

- the optimal number of workers model
- the load balancing model

We will specify the tunlets taking into account these models to tune applications created by using the framework Master/Worker [36] associated with MATE [10, 11], presented in Section 2.3. In order to generate the tunlets we follow the methodology presented in Chapter 4. Even though in this work we want to evaluate the usability of the proposed methodology and the specification language, we present in addition some results obtained when the application is executing by itself and when it is executed under MATE.

Note that these use cases are not just examples of the language usage. In fact, the two automatically developed tunlets will provide MATE with two new tunlets, available to straightforwardly tune applications developed with the Master/Worker Framework. For applications developed following another methodology, the tunlets can be reinterpreted according to the application.

5.1 Optimal Number of Workers

In Master/Worker applications it is crucial to study the application performance according to the quantity of workers used to process the tasks, because it is one of the major performance problems in this programming scheme. When there are not enough worker processes, the master process distributes the data and becomes idle as it waits for results. On the other hand, if there are too many workers, the amount of data is divided into small pieces and the communications saturate the system.

5.1.1 Providing a Performance Model

The model provided to develop the tunlet is a pre-existing performance model. We present the general aspects of it, but additional details can be obtained from [14].

Terminology

In the following, we present the terminology used to identify the different parameters of the performance model:

- tl = network latency, in milliseconds (ms)
- λ = sending a byte cost (bandwidth inverse relation), in $\frac{ms}{byte}$
- v_i = size of tasks sent to each worker i, in bytes
- v_m= size of the answer send back to the master for each worker (bytes)
- V= total data volume($\sum(v_i + v_m)$), in bytes

- n= current number of workers
- tc_i= computing time of the worker i, in ms
- Tc= total computing time ($\sum(tc_i)$), in ms
- Tt= total iteration time, in ms
- $Nopt$= number of workers needed to minimize the execution time

Tt is the magnitude we want to minimize, and to do that we should use $Nopt$ workers.

Performance Functions

The following expression indicates how to calculate the number of workers suitable to improve the application performance:

$$Nopt = \sqrt{\frac{\lambda V + Tc}{tl}}$$

where $Nopt$ represents the number of workers needed to minimize the execution time. This expression was obtained by deriving the expression that models the execution time of an iteration, in order to minimize it. Such expression is defined in function of computing time and communication time, which is influenced by the latency and bandwidth (more details can be obtained from [14]).

Performance Parameters

To calculate Tt a set of parameters should be measured. For each one of they, the model indicates where and when should be measured, to allow evaluating the performance functions:

- tl (network latency, in milliseconds) and λ (sending a byte cost - bandwidth inverse relation- in $\frac{ms}{byte}$). They must be calculated at the beginning of the execution and should be periodically updated to allow the adaptation of the system to the network load conditions.
- V is the total data volume($\sum(v_i + v_m)$) expressed in bytes, where:

  - v_i (size of tasks sent to each worker$_i$, in bytes) must be captured when master sends tasks to the workers.
  - v_m (size of the answer send back to the master for each worker, in bytes) must be captured when master receives answers from the workers.

- Tc is the total computing time ($\sum(tc_i)$), in ms, where:
  - tc_i(computing time of the worker i, in ms). Each worker computing time is needed to calculate the total computing time (Tc).

5.1.2 Interpreting the Performance Model

As indicated in Section 4.3.2, even though the performance expressions could not be very intuitive without the deep study of the model, it is enough to understand the meaning of each parameter in order to correctly identify what are the entities in the application which embody such parameters. In the previous section we explained the meaning of each performance parameter; this is why in this analysis we do not include the *Understanding the Performance Model* step of the methodology. In this section, those parameters will be interpreted according to the framework under study.

Identifying the variables and values

We need to interpret V (v_m and v_i), Tc (tc_i), tl, λ and $Nopt$, by identifying the events to be caught and the information associated to them. First of all we have to identify the variables and values in the application -in this case in the framework- which conform the information required to evaluate the performance parameters. We start by analyzing the variables and values which can be straightforwardly identified. However, as we progress in the definition of the tunlet, new variables to take into account could appear.

- The value of v_i can be obtained from the variable *numtuples* which indicates (in the sending action of the master process) the number of tasks sent to the worker i. In addition, we need the variable called $TheWorkUnitBytes$ -which indicates the size in bytes of each task- to multiply by the cumulative addition of every *numtuples*.

- The value of v_m can be obtained from the variable $nbytes$, used by the master process to indicate the size of the answer received from each worker.

- ct_i has to be calculated for each worker according to the instants in which the worker starts and finishes the computing phase. In this case, we do not need any variable or value from the framework, due to we only need to know some timestamps.

- tl is considered as fixed ($tl = 1000ms$)

- λ has to be calculated as the division between the communication time of the iteration and the volume of data interchanged with a worker. The way to obtain the volumes of data was explained above. The question is: how to calculate the communication time of an iteration? Such time can be calculated by capturing:

  - the instant of the last receive (lr) of the master
  - the instant of the first send (fs) of the master
  - the computing time (i.e. tc_i) of the worker which send the results at last .

  Then, the communication time can be calculated as $(lr - fs) - tc_i$. Observe that in this case we do not need to extract information from variables, due to λ depend on some timestamps and tc_i. However, we need to identify what is the "first" and what is the "last" workers. Then, we have to consider two new variables:

  - $workerTID$, which is used by the master to know what is the worker it is sending data to, and
  - $workerTIDr$, used by the master to know what the answering worker is.

  In addition, due to we need tc_i of the last worker, when capturing tc_i we need to know which is such "i". Thus, we need to consider:

  - $myTID$, the variable used by the worker to indicate its tid. However, we can dispense with this variable, due to events have a default attribute *id* in order to indicate the process in which was caught.

- The value of $Nopt$ has to be assigned to the variable in the framework which controls the number of workers in each iteration. The framework provides the $NOptWorkers$ variable in order to introduce the changes, while the variable used through the iteration to control the current number of workers is nw.

Note that V and Tc do no need any variable value, due to they are calculated in function of v_i and v_m, and tc_i, respectively.

From the general point of view of the application, and in a more deep analysis, when an event is caught we need to identify what is the current iteration to associate the information to the correspondent iteration. The framework uses two variables to indicate the current iteration:

- *iteration* in the code of the master, and
- *CurrentIteration* in the code of the worker.

In the Table 5.1 we present the main information related to each involved variable, which will be used to formalize the abstractions in the corresponding specification:

Identifying the events

The next step consists in to determine how, where and when to capture the variables and values previously enumerated. Note that the names used to call the events could be any names; which we use try to be mnemonic. In general, "*M*" stands for *master* and "*W*" stands for *worker*. In Table 5.2 we summarize the main information related to each event.

The *MSendsTaskW* captures each execution of the sending action and the number of tasks sent to each worker, while the *WRepliesM* event collects the volume of data received by the master each time a worker sends the results. *WStartsTask* and *WFinishesTask* are used to capture the instants

Variable	*Type*	*Actor*
numtuples	int	master
TheWorkUnitBytes	int	master
nbytes	int	master
NOptWorkers	int	master
nw	int	master
workerTID	int	master
workerTIDr	int	master
iteration	int	master
CurrentIteration	int	worker

Table 5.1: Information about the variables

delimiting the data processing in each worker. In the case of *WFinishesTask*, the considered place to be caught is the entry of $\_W\_SendAll$ method, due to among the end of $\_ReceiveEffective$ and such place, some final computing is executed in the worker process. In addition to the events necessary to catch the variables and values previously depicted, we have to define the events which indicate the start and the end of an iteration, in order to be able of evaluating the performance expression. In the framework, each iteration is delimited in the code of the master. The loop starts by executing a function to divide the data and send the tasks, and finishes with a function to receive answers and make the final treatment of the results. Thus, we can consider such entry and exit, respectively, in order to catch the iteration. Note that the attributes associated to each event are visible to the code of the actor where the event is captured (compare Tables 5.1 and 5.2).

5.1.3 Identifying the Actors

Due to we are working with a Master/Worker framework, there are clearly two kinds of processes: *master* and *worker*. Due to for each one of them we need collect some specific information (for the workers we need the computing time and for the master we need the volume of data and communication time) we will need to instrument each kind of process in a different manner. Then, we have two different actors, and for each one of them we need to

Event	*Attributes*	*Actor*	*Method*	*Place*
MSendsTaskW	numtuples workerTID	master	_DMM_SendTask	exit
WRepliesM	nbytes workerTIDr	master	_DMM_ReceiveEffective	exit
WStartsTask	timestamp id	worker	_W_DoWorks	entry
WFinishesTask	timestamp	worker	_W_SendAll	entry
IterationStarts	iteration TheWorkUnitBytes nw	master	_M_SendIteration	entry
IterationFinishes	iteration	master	_M_ReceiveIteration	exit

Table 5.2: Information about the events

identify some information:

Master

- id: master
- min:1
- max: 1
- class: _CMaster, _MyMaster
- executable file: /home/paola/pvm3/bin/LINUX/master
- completion condition: the event "IterationFinishes" happened
- attributes: the instants of the first send (fs) and the last receive (lr), and the worker who answered at last (i.e. the value of *tid* of the worker registered in lr).

Worker

- id: worker

- min:1
- max: 28
- class: _CWorker, _MyWorker
- executable file: /home/paola/pvm3/bin/LINUX/worker
- completion condition: the processing of data has been finished.
- attributes: the instants in which the computing starts and when finishes, to calculate the computing time (tc_i); the volume of data received (v_i) and sent (v_m) and the instants in which where received and sent. Strictly, those two instants are the instant in which the master sends a task to worker$_i$ and when receives the answer. However, even though they are instants caught in the master process, due to the information is closer to the workers, it is stored in the worker structures. In other case, the master should include an array to store the times corresponding to each worker.

5.1.4 Tunlet Specification

In the previous sections, we defined the main abstractions involved in the definition of the tunlet. In this section, we present how to formalize such abstractions. We will explain the main aspects to consider and provide some examples on how to formalize each kind of entity. The complete specification can be consulted in Appendix B.

Measure points

VARIABLES AND VALUES

The variables and values able to be "manipulated", both to obtain their values and/or change it, have to be declared in this section of the specification. As can be appreciated in Appendix B, for each variable we formalize the information provided in the previous sections, and we provide some additional properties associated to each variable. As an example, consider the *numtuples* variable:

```
variable
   id: numtuples
   comment:/*the amount of tuples sent to a worker*/
   source: asVarValue
   type: int
   actorId: master
endvariable
```

On the one hand, in addition to the information predetermined (i.e. the name, the type and the actor), in the *comment* property we provide a brief description of the semantic of the variable in the framework. This description could be useful in future uses of the tunlet. On the other hand, we include the *source* property which indicates the nature of the entity, in this case indicated by *asVarValue*, due to *numtuples* is a variable. Remind what mentioned in Chapter 4: this property could take some value among the following ones:

```
  {asFuncParamValue, asVarValue, asFuncReturnValue,
   asConstValue, asFuncParamPointerValue}
```

EVENTS

In this section we formalize the description of the events. Consider as an example the $MSendsTaskW$ event:

```
event
   id: MSendsTaskW
   actorId: master
   controliter: no
   utility: always
   method:_DMM_SendTask
   class:_CDtaMngM
   place: exit
 ATTRS:
   id:workerTID
   id:numtuples
endevent
```

We included some additional properties to the presented in Table 5.2, such as:

- *class*, which in combination with the *method* property is used to locate the function required. If the application is not following an object oriented paradigm, this property takes the value *none*.
- *controliter*, used to indicate if the event is controlling the start or the end of an iteration, or if the reception of this event triggers the evaluation of the performance model. Then, the possible values are *begin*, *end* and *eval* respectively. In other case, the value is *no*, such as in the example, due to the sending of a mesage does not mean the start or end of an iteration nor the need for evaluating the performance model, because the processing and receiving information remains to be collected.
- *utility* is a property used to indicate if the event is used along all the execution or if it is an event to add or remove in case it is necessary. Then, the possible values for this property are *always*, *addable* and *removable* respectively. In this example, the event has to be caught in every iteration.

The attributes associated to the event are enumerated after the *ATTRS* word.

ACTORS

In this section, the actors of the programming model are depicted. Consider the master:

```
actor
   id: master
   min: 1
   max: 1
   completion: /#master.comp==1#/
   class:_CMaster, _MyMaster
   exe: /home/paola/pvm3/bin/LINUX/master
```

```
ATTRS:
  id:firstSend
  comment:/*timestamp of the first send*/
  type: double
  inic:/#firstSend=0.0;#/
  depinic:none
  value:/#   if( (MSendsTaskW.timestamp  <
                  master[MSendsTaskW.id].firstSend) ||
                 (master[MSendsTaskW.id].firstSend==0.0)
               )
             {
                master[MSendsTaskW.id].firstSend=
                                  MSendsTaskW.timestamp;
             }
       #/
  cum:false
  dependency: MSendsTaskW

  ...

endactor
```

In this case, we mainly formalized each property. For this example we present just one attribute even though this actor has several of them. The attributes of actors allow for storing the information related to each actor until all the required information is available to evaluate or update the performance parameters. Some of the properties of such attributes are the same as for variables and values (i.e. *name*, *type* and *comment*); the additional ones are the following:

- *inic*, used to determine the initial value of the attribute.
- *value*, indicates how to determine the value of the attribute. From the example, remind each event has three default attributes: *timestamp*, *id* and *task*, indicating the instant, the process id and the task in which

the event was caught. In addition, remember that the data associated to each actor is accessed using iterators and selectors, such as in every data structure.

- *dependency*, which indicates when the attribute can be updated, i.e, if it depends on the value of other entities. In the example, *firstSend* can be calculated when the *MSendsTaskW* event is received.
- *depinic* indicates the dependences of *inic* property. In this case, due to *firstSend* in initialized in 0.0 it has *none* dependency.

ITERATION INFORMATION

In Section 5.1.2 we identified as necessary the variables *iteration* and *CurrentIteration* in order to associate the information to the correspondig iteration data structure. In the specification, we included several attributes in order to store information inherent in each iteration. In the case of *StartedIteration* takes the value carried by *IterationStarts.iteration*. In the case of *tuplesize*, we need to know the size of each task in order to calculate the performance parameter V. Finally, *iterCommTime* takes the value from $(fs - lr) - ct_i$, which is used to calculate the value of λ. We define each attribute of this section with the same description as the attributes of actors.

MODEL PARAMETERS

Each entity declared in this section is described with the same properties as the attributes of actors. Consider for example *vm*:

```
id:vm
comment:/*answers volume*/
type: int
inic:/#vm=0;#/
depinic:none
value:/# vm=0;
         for(int i=0;i<n;i++)
         {
           vm+=worker[i].replysize;
```

```
            }
          #/
   cum:true
   dependency: none
```

In this case, the value of *vm* is calculated as the addition of the *replysize* property of each worker. *replysize* takes its value from *WRepliesM.nbytes*. In the particular case of model parameters, we specify a dependency just when a parameter is depending on another one. But we do not consider external dependences (this is dependences on events) due to we assume the performance parameters are evaluated when every event of the iteration was received and processed.

Performance Functions

In this performance model we have just one performance function. Then, we just implemented the function using the library functions available for C/C++. In this case, we use *math.h*.

Tuning Points

NOptWorkers is the variable whose value has to change in order to tune the number of workers used to process the data. We define the tuning point as follows:

```
point
   id: NOptWorkers
   value:/#pf();#/
   kind: SetVariableValue
   syncfunction=0
   syncplace=0
   cond:/#NOptWorkers > iter.GetNum_worker()+2 ||
          NOptWorkers < iter.GetNum_worker()-2
        #/
endpoint
```

The information associated to the *NOptWorkers* variable is included in the *MEASURE POINTS* section. Here, it acts as a reference. The *syncfunction* and *syncplace* properties are used to indicate, if needed, where should be introduced a breakpoint in order to tune the variable and avoid inconsistencies. In this example, there is needed no breakpoint, due to the value of *NOptWorkers* is considered only at the beginning of each iteration, then we declare the values as 0. The *cond* property indicates when the tuning has to be applied. In this case, we considered that the number of workers has to changed when the difference among the current amount and the optimal is bigger than 2. The value of *NOptWorkers* is calculated using the performance function *pf()*. The kind of tuning action to perform is indicated in the *kind* property.

5.1.5 Generated Tunlet

The automatically generated tunlet implements the logic provided by the performance model according to the application. In particular, due to the Analysis phase has a previous stage of collection and preprocessing of the data in the Collector-Preprocessor machines, the evaluation (i.e. the calculations to update the value) of each actor attribute, iteration attribute or performance parameter is made as follows:

- **Collector-Preprocessor**: Collects the events related to the workers (`WStartsTask` and `WFinishesTask`), and the attributes of such events are stored in the corresponding worker structures. When every worker finished the iteration, the Collector-Preprocessor performs the final calculations involving the local data. In this case, calculates the partial Tc by adding every tc_i (at the same time, tc_i is calculated in function of the instants in which the computing started and finished). Then, the partial Tc and every tc_i are sent to the Analyzer, in order to complete the missing information necessary to evaluate the performance model.

- **Global Analyzer**: Collects the events caught in the master process code (`MSendsTaskW`, `WRepliesM`, `IterationStarts`, `IterationFinishes`). The attributes associated to each event are stored. Concurrently, the

Analyzer receives from Collector-Preprocessor the summary of the collected information and reclassifies it. In particular, the partial Tcs are accumulated in the global Tc and each tc_i is stored as corresponds. Thus, the Analyzer can obtain the computing time (tc_i) corresponding to the worker which answered at last, needed to calculate λ. Once every performance parameter has been evaluated or updated, the performance model can be evaluated.

5.1.6 Experiments

In this section we want to validate the usefulness of the tunlet automatically generated. We compare the execution time of the application when executed under the automatically generated tunlet and when executed by itself in different fixed number of workers (1, 2, 4, 8, 16, 25). To conduct the experiments, we selected a 2D N-Body created by using the Master/Worker framework [36]. Experiments were conducted on a homogeneous cluster with the following configuration:

- *Processor PENTIUM IV 3.0 Ghz (Fedora Core 4)*
- *1 GB DDR-SDRAM 400 Mhz*
- *Ethernet card Broadcom NetXtreme Gigabit*

We created certain load patterns, so that we can introduce and modify certain external load to simulate the system's timesharing. Each experiment was performed many times and the average of the execution time for the application was calculated. Results are shown in Table 5.3 and Figure 5.1.

Number workers	1	2	4	8	16	25
Execution Time	13,37	7,42	4,06	2,18	2,58	3,34
NBody + MATE	Starting with 1 worker and then tuning					
Execution Time	1,42					

Table 5.3: Execution time of NBody considering different fixed numbers or workers and NBody under MATE (in minutes)

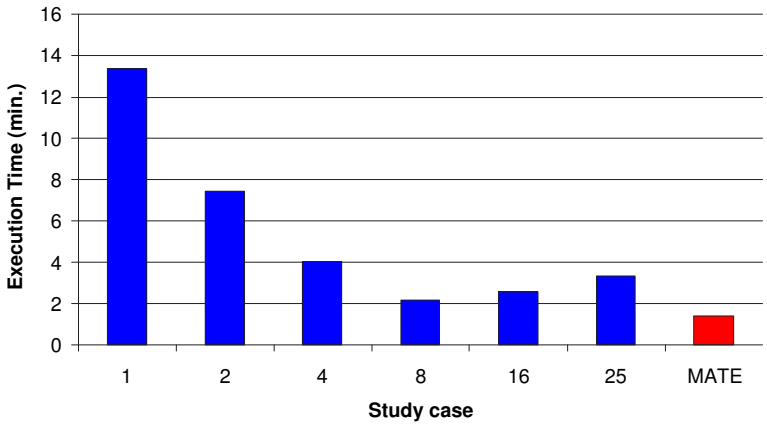

Figure 5.1: Execution times of NBody when considering the number of workers

As shows the table and the figure, when the application is executed under MATE it obtains the best execution time. In the cases in which the number of workers is fixed, the execution time is affected or degraded by the overload of the workers or by the communication time. The first reason can be observed when 1, 2 or 4 workers where used. The second reason can be observed in the case of 16 and 25 workers. If we consider the best execution time obtained with a fixed number of workers, i.e. when the application was executed with 8 workers, we can observe that MATE allows for reducing the total execution time about 30 per cent. In addition, the advantage of using MATE, is that the dedicated resources to the application are determined in each instant according to the current conditions of the system. In consequence, the resources are used just when they are necessary.

5.2 Load Balancing

Imbalance in parallel systems could be caused by heterogeneity of processors, operating system interference or irregularity of the tasks assigned to a processor. In general, the load balancing techniques try to compensate the load imbalances by assigning more work to processors that finish earlier their work. Factoring is a load balancing strategy which divides the total work in batches. Each batch has as many chunks as processors are executing, every chunk containing the same amount of tasks.

5.2.1 Providing a Performance Model

As in the previous example, the model provided to develop the tunlet is a pre-existing performance model. We present the general aspects of it, but additional details can be obtained from [15].

Terminology

The terms involved in the performance model are the following:

- *iteration* = an iteration of the application, in which a data set has to be processed.
- N = number of tasks to be processed in an iteration
- *batch* = each one of the parts in which N is divided to implement the factoring algorithm. Each batch constitutes a subset of tasks.
- *subiteration* = the processing of a batch of tasks
- x_i = portion of tasks to be processed in the subiteration i
- F_i = amount of tasks assigned to each chunk of subiteration i
- R_i = remaining amount of tasks (for next subiterations) when subiteration i is been executed.
- C = task processing time ($ms/task$)
- $\mu(C)$ = mean of task processing time
- $\sigma(C)$ = standard deviation of task processing time
- P = number of workers

Performance Functions

The following expression indicates how to calculate the amount of tasks to be assigned in the subiteration to obtain the best performance, according to the current conditions of the system:

$$F_i = \begin{cases} \frac{N}{x_i P} & \text{, if } i = 0 \\ \\ \frac{R_i}{x_i P} & \text{, otherwise} \end{cases}$$

The differences between both definitions reside in the fact that in the first subiteration (when $i = 0$) the workers are synchronized, waiting for data to be processed, whilst in the remaining subiterations the availability of the workers depends on the speed the previous batch was processed with. For more details, consult [15].

Performance Parameters

In order to calculate the value of F_i the following magnitudes have to be considered as indicated:

- C (task processing time) necessary to calculate $\mu(C)$ and $\sigma(C)$. Has to be measured in every iteration.
- $\mu(C)$ necessary to calculate F_i
- $\lambda(C)$ necessary to calculate F_i
- The successive x_i of an iteration have to be calculated at the start of the iteration in order to establish the size of the relating F_i. It should be calculated as follows:

$$x_i = \begin{cases} \frac{\mu(C)+\sigma(C)\sqrt{\frac{P}{2}}}{\mu(C)} & \text{, if } i = 0 \\ \\ \frac{2\mu(C)+\sigma(C)\sqrt{\frac{P}{2}}}{\mu(C)} & \text{, otherwise} \end{cases}$$

- P has to be measured when the application starts, and should be periodically updated in case the number of workers could change along the execution. In our case, we consider P as fixed.

5.2.2 Interpreting the Performance Model

Such as in use case 5.1, the performance model was explained in the previous section, then in this one we define the abstractions involved in the definition of the tunlet.

Identifying the variables and values

For this model, we need to interpret N, x_i, F_i, R_i, C and P. Then, we will identify the variables and values in the application -the framework- which embody the parameters.

- C has to be calculated as the average of the computing time wasted by a worker to process each one of the received tasks. In order to reduce the amount of events, in place of capturing the time wasted on executing each separate task, we can calculate $C = average(tc_i/NroTasks)$, where:
  - tc_i represents the computing time of the worker i through the iteration, which can be calculated by catching the instants in which the worker starts and finishes the computing phase, as in previous use case. However, in this case the computing time is the addition of every batch computing time (in the previous use case, each worker processed only one -bigger- batch along each iteration).
  - $NroTasks$ is a variable in the worker process which indicates how many task have been received.
  - $TheNumTuples$ and $TheWorkUnitBytes$ are variables in the master process which indicates the total amount of tasks of the iteration and the size of each task in bytes, respectively. In combination, these variables allows for determining N and initialize R_i.

  As in the previous use case, there are two variables necessary in order to identify the worker which is receiving or sending the data:

  - *workerTID*, which is used by the master to know what is the worker it is sending data to, and
  - *workerTIDr*, which is used by the master to know what is the worker which is answering.

- The value of F_i has to be assigned to the variable in the framework which controls the size of the chunk. The framework provides the *globalSizeChunk* variable in order to introduce the changes.

- The value of P can be obtained from the *nw* variable, used by the master process to control the amount of workers along the iteration.

Note that $\mu(C)$, $\sigma(C)$, x_i and F_i do not need any variable value, due to they are calculated in function of the other parameters.

Variable	*Type*	*Actor*
NroTasks	int	worker
TheWorkUnitBytes	int	master
TheNumTuples	int	master
nw	int	master
globalSizeChunk	int	master
workerTID	int	master
workerTIDr	int	master
iteration	int	master
CurrentIteration	int	worker

Table 5.4: Information about the variables

As in the previous use case, from the general point of view of the application, when an event is received we need to identify what is the iteration it belongs to in order to associate the information to the correspondent iteration. As mentioned before, we can use the variables used by the framework to indicate the current iteration:

- *iteration* in the code of the master, and

- *CurrentIteration* in the code of the worker.

In the Table 5.4 we present the main information related to each involved variable, which will be used to formalize the abstractions in the corresponding specification:

Identifying the events

The next step consists in to determine how, where and when to capture the variables and values previously enumerated. For familiarity, we used the same names as in the previous use case. In Table 5.5 we summarize the main information related to each event.

Event	*Attributes*	*Actor*	*Method*	*Place*
MSendsTaskW	workerTID	master	_DMM_SendTask	exit
WRepliesM	workerTIDr	master	_DMM_ReceiveEffective	exit
WStartsTask	timestamp CurrentIteration NroTasks	worker	_W_DoWorks	entry
WFinishesTask	timestamp CurrentIteration	worker	_W_SendAll	entry
IterationStarts	iteration TheWorkUnitBytes TheNumTuples nw	master	_M_SendIteration	entry
IterationFinishes	iteration	master	_M_ReceiveIteration	exit

Table 5.5: Information about the events

As in the previous use case, we define *IterationStarts* and *IterationFinishes*, due to are necessary to indicate the start and the end of an iteration, in order to be able of evaluating the performance expression. In general, the information collected to complete the performance parameters of the model is very similar to the information collected to the model presented in the previous use case. The difference resides in the manner in which the information is interpreted and used. A particular difference is the use of the *NroTasks* variable. The model under consideration requires the calculation of the average task processing time, whilst in the previous use case we worked

at volume of data (in bytes) level.

5.2.3 Identifying the Actors

The underlying programming model has two actors: master and worker. Then, we define both actors and their attributes:

Master

- id: master
- min: 1
- max: 1
- class: _CMaster, _MyMaster
- executable file: /home/paola/pvm3/bin/LINUX/master
- completion condition: the event "IterationFinishes" happened
- attributes: none

Worker

- id: worker
- min:1
- max: 25
- class: _CWorker, _MyWorker
- executable file: /home/paola/pvm3/bin/LINUX/worker
- completion condition: the processing of data has been finished.
- attributes: the instants in which the computing starts and when finishes to calculate the computing time (tc_i), the amount of tasks received, C (the average task processing time).

5.2.4 Tunlet Specification

In the previous sections, we defined the main abstractions involved in the definition of the tunlet. Due to we explained the general formalization of such abstractions in the previous use case, in this section we present an example of the usage of an auxiliary performance parameter, used to make possible the precalculation (at CPs level) of some cumulative operation. The complete formalized specification can be consulted in Appendix C.

Consider $\mu(C)$ (the mean of task processing time). It can be calculated as follows:

$$\mu(C) = \frac{\sum C_i}{P}$$

where C_i represents the task processing time of worker i, $0 \leq i \leq P$. Therefore, in order to calculate the value of $\mu(C)$, have to be obtained or calculated in advance the values of P and every C_i. If we now consider how the information about the application behaviour is collected according to the previous analysis and the formalized specification (Appendix C), we find the following:

- P is obtained from *IterationStarts.nw*, which is captured in the master process.
- Each C_i is calculated as $tc_i/numTaskRecv$, where at the same time tc_i is calculated according to the *WFinishesTask.timestamp* event and the *WStartsTask.timestamp* event and $numTaskRecv$ is obtained from *WStartsTask.NroTasks*. Obviously, both tc_i and $numTaskRecv$ are obtained from the worker process (see Section 5.2.2 and Table 5.5 for clarification).

According to the current implementation of distributed-hierarchical Analyzer and given that P and C_i are obtained in different actors, they are managed at different levels: CPs collect the incoming events from workers and Global Analyzer collects the incoming events from Master.

If we specify $\mu(C)$ as:

```
id:MC
comment: /*mean of C*/
type:double
inic: /#MC=0.0;#/
depinic:none
value: /#for(int i=0;i<n;i++)
          {MC+=worker[i].C;}
          MC=MC/N #/
cum:true
dependency:none
```

even though we declare that the calculus of the value of `MC` includes a cumulative operation (`cum:true`), the operation cannot be dismembered given that the Tunlet Generator operates as follows:

- The Tunlet Generator interprets `cum:true` as a possible operation to be executed by CPs.
- The actors involved in the necessary information are considered.
- If the information depends only on one actor, the value can be calculated at CP level if the actor is the worker, or at Global Analyzer level if the actor is the master. Such value can be calculated as soon as every parameter involved in the calculation is available.
- If the information depends on different actors, the calculation of the value is deferred until Global Analyzer collected all the involved information.

In order to take profit from the distributed-hierarchical collecting-preprocessing approach, we can declare an auxiliary performance parameter to allow the CPs for calculating the partial cumulative addition of the local C_i; in this manner, Global Analyzer does not need every C_i. Then, we specify:

```
id:Ct
comment: /*partial cumulative addition of Ci*/
type:double
```

```
inic: /#Ct=0;#/
depinic:none
value: /#for(int i=0;i<n;i++)Ct+=worker[i].C;#/
cum:true
dependency:none

id:MC
comment: /*mean of C*/
type:double
inic: /#MC=0;#/
depinic:none
value: /#MC=Ct/N;#/
cum:false
dependency:Ct
```

In this way, `Ct` is calculated at CP level, due to the value involves only workers information. Even though Global Analyzer has to calculate the value of `MC`, we are reducing the amount of operations to be executed by the Global Analyzer and we exempt the CP from send every C_i, which reduces the size of the message sent to the Global Analyzer.

As explained before, Global Analyzer does not need C to calculate `MC` when CP calculates the partial Ct. However, as the translation continues, the Tunlet Generator detects that $\sigma(C)$ depends on master and worker. Unfortunately, in this case the calculation cannot be dismembered given that $\mu(C)$ (`MC`) has to be determined beforehand. In consequence, every C has to be sent to the Global Analyzer to calculate $\sigma(C)$. However, the reduction of the amount of operations done by the Global Analyzer (commented in the previous paragraph) is still valid.

5.2.5 Generated Tunlet

The logic provided by the performance model studied in the previous analysis was automatically transformed in a tunlet. The functioning of each stage of Analysis phase was determined as follows:

- **Collector-Preprocessor**: Collects the events related to the workers

(`WStartsTask` and `WFinishesTask`), and stores the information carried by them in the corresponding worker data structure. When every worker finished the iteration, the Collector-Preprocessor performs the final calculations involving the local data. In this case, calculates C for each worker as the division between tc_i and $numTasksRecv$, where tc_i is calculated in function of the instants in which the computing started and finished and $numTasksRecv$ takes its value from $WStartsTask.NroTasks$. In addition, each CP calculates the partial Ct (used to calculate $\mu(C)$, see Appendix C) by adding every C. Then, every C and the partial cumulative addition Ct are sent to the Analyzer, in order to complete the missing information necessary to evaluate the performance model.

- **Global Analyzer**: Collects the events caught in the master process code (`MSendsTaskW`, `WRepliesM`, `IterationStarts`, `IterationFinishes`). The attributes associated to each event are stored. Concurrently, the Analyzer receives from Collector-Preprocessors the average task computing time C of each worker, and the partial cumulative addition of them (the partial Ct) of the workers, which is accumulated in the global Ct. Thus, the Analyzer can obtain calculate the the average task computing time and the standard deviation. Once every performance parameter has been evaluated or updated, the performance model can be evaluated.

5.2.6 Experiments

To conduct the experiments, we used the same 2D N-Body application and the same platform as in the previous use case. In this section we compare the execution time of the application in three different scenarios:

- the application was executed by itself in different fixed number of workers (1, 2, 4, 8, 16, 25) in a dedicated environment (i.e. without any additional external load)

- the application was executed in a non-dedicated environment. In this case we injected a controlled variable load.

- the application was executed under the automatically generated tunlet

Each experiment was performed many times and the average of the execution time for the application was calculated. Results are shown in Table 5.6 and Figure 5.2.

Number workers	1	2	4	8	16	25
N-Body	10,14	5,46	3,38	1,48	2,31	2,37
N-Body + variable load	13,37	7,42	4,06	2,18	2,58	3,34
N-Body + MATE + variable load	-	6,01	3,41	1,51	2,48	2,52

Table 5.6: Execution time of NBody considering different fixed numbers of workers with and without extra load, and NBody under MATE (in minutes)

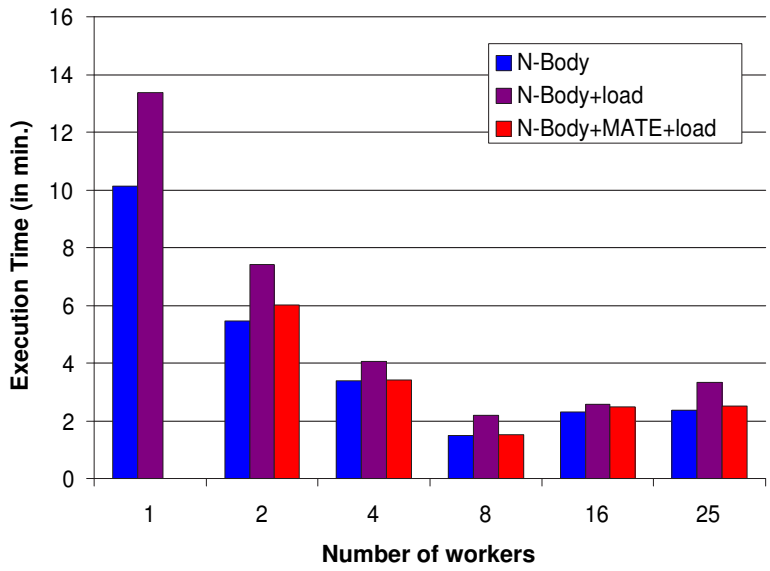

Figure 5.2: Execution times of NBody when considering load balancing

As shows the table and the figure, the execution time of the application increases when the external controlled load is injected. However, the imbalance is in general corrected when the application is executed under MATE, due to MATE detects the changes in the conditions of the system and adapts the factors used to distribute the work.

Chapter 6

Conclusions

> "¿Qué se puede hacer en ochenta años? Probablemente, empezar a darse cuenta de cómo habría que vivir y cuáles son las tres o cuatro cosas que valen la pena. Un programa honesto requiere ochocientos años. Los primeros cien serían dedicados a los juegos propios de la edad, dirigidos por ayos de quinientos años; a los cuatrocientos años, terminada la educación superior, se podría hacer algo de provecho; el casamiento no debería hacerse antes de los quinientos; los últimos cien años de vida podrían dedicarse a la sabiduría. Y al cabo de los ochocientos años quizá se empezase a saber cómo habría que vivir y cuáles son las tres o cuatro cosas que valen la pena. Un programa honesto requiere ocho mil años. Etcétera. "
>
> *Uno y el Universo*, Ernesto Sábato

THIS chapter presents the conclusions and results obtained from our work. The chapter also describes the possible open lines that can be undertaken in the future in order to continue the research on dynamic and automatic tuning of parallel applications.

6.1 Conclusions

In this work we have treated an important aspect of high performance computing: the tuning process of parallel applications.

The main purpose of existing works in the field is to provide means to detect and solve performance problems. Each tool provides support for some or every step in the tuning process. However, each one obeys to a particular

approach and has a particular philosophy. In Chapter 1 we discussed the applicability and constraints of them. In particular, in this work we have been focussed in MATE (Monitoring, Analysis and Tuning Environment) which provides dynamic and automatic tuning of parallel applications. This environment was described in Chapter 2. Part of the description, development and results of can be found in:

> MATE: Monitoring, Analysis and Tuning Environment for Parallel/Distributed Applications. A. Morajko, P. Caymes-Scutari, T. Margalef, E. Luque. Concurrency and Computation: Practice and Experience. 2005. *Accepted.*

> Automatic Tuning of Data Distribution Using Factoring in Master/Worker Applications. A. Morajko, P. Caymes, T. Margalef, E. Luque. 5th Internacional Conference in Computational Science (ICCS2005). Part III (LNCS 3515), p. 132-139. 22-25 May 2005. Atlanta, GA, United States.

In this work we have proposed and developed two different extensions of MATE in order to expand, improve and facilitate its usability. On the one hand we worked on the scalability of MATE and on the other hand we worked in making MATE transparent for the users.

With regard to the scalability of MATE, we proposed a novel approach to execute the analysis process: the *Distributed-Hierarchical Collecting-Preprocessing Approach*. Until now, MATE has been executing the analysis phase in a centralized manner, which turned in a bottleneck as the number of machines (and events) in the application was in increase. We studied the different possibilities or approaches to overcome such a problem. Both distributed and hierarchical approaches present constraints to be applied, due to they require to introduce modifications in the application or/and in the performance model. This would entail additional work for the user, who would be responsible for adapting the performance model or the application. These actions sometimes could provoke inconsistencies, specially when we are talking about non expert users. Then, on the one hand we proposed to distribute what can be distributed independently from the performance

model and from the application: the collection of events; on the other hand we proposed decentralize the processing of the information carried by the events, then we proposed to define a hierarchy over the calculations which can be dismembered, such as cumulative or comparative operations.

We developed and coded a first version of this new approach in analysis process. We have a set of collector-preprocessors (CPs) and a Global Analyzer. Each CP executes in an independent machine and is responsible for collecting the incoming events from a predetermined set of machines. The information associated to the events is preprocessed and when all the information of an iteration is completed, the CP condenses the information needed by Global Analyzer and send it. Global Analyzer executes in another independent machine; it receives a small portion of events and the data sent by the CPs. Thus, the Global analyzer is responsible for evaluating the performance model and requiring for tuning actions.

We showed the distributed-hierarchical collecting-preprocessing approach allows for overcoming the bottlenecks presented by the full centralized approach. We improved the synchronization of the analysis process with the execution of the application in several manners: first of all, due to the Global Analyzer is receiving just a part of the total volume of events, we augment the probability of detecting and processing each event as soon as received. Particularly interesting is the detection of the start and the end of an iteration, owing to them indicates all the information belonging to such an iteration has been already caught, and the Global Analyzer will be able of evaluating the performance model as soon as every CP sends the preprocessed information. That is precisely the second improvement: the experiments showed that the Global Analyzer has a minimal delay to start the updating or calculation of performance parameters and the evaluation of the performance model, due to it has all the preprocessed information available when the end of the iteration is reached and detected. Another reduction of time is in the updating/calculation of performance parameters, due to a part of the calculations has been done in CPs. However, this reduction is proportional to the volume of calculations dependent on preprocessed data.

Relating to making MATE transparent to the user, we proposed and developed a methodology to automatically generate tunlets from specifications.

Until now, if users wanted to use MATE to tune their applications, they had to program the corresponding tunlet considering the requirements and implementation details of MATE or they had to develop their applications using the Master/Worker Framework associated to MATE and tune the number of workers. The details and results of the developments to constitute the development and tuning environment (framework + MATE) can be found in:

> Development and Tuning Framework of Master/Worker Applications. A. Morajko, E. César, P. Caymes-Scutari, J. Mesa, G. Costa, T.Margalef, J. Sorribes, E.Luque. Journal of Computer Science & Technology (JCS&T) - October 2005. Invited Paper. Vol. 5 N. 3. P. 115-112. ISSN:1666-6038. October 2005. Argentina.

> Entorno de Desarrollo y Sintonización de Aplicaciones Master/Worker. P. Caymes-Scutari, A. Morajko, E. César, J. Mesa, G. Costa, T.Margalef, J. Sorribes, E.Luque. IX Congreso Argentino de Ciencias de la Computación CACIC 2005. Electronic Proceedings of the conference. 17/10/05 - 21/10/05. Concordia, Argentina.

> Automatic Tuning of Master/Worker Applications. A. Morajko, E. César, P. Caymes-Scutari, T. Margalef, J. Sorribes, E. Luque. Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference. LNCS 3648, p. 95-103. 30/08/05 - 02/09/05 Lisboa, Portugal.

In addition, such developments where presented in some workshops:

> Development and Tuning Environment of Master/Worker applications. P. Caymes-Scutari, A. Morajko, G. Costa, E. César, T.Margalef, J. Sorribes, E.Luque. Automatic Performance Analysis. Dagstuhl Seminar N 05501. Schloss Dagstuhl - International Conference and Research Center for Computer Science. 12-16 December 2005. Dagstuhl, Germany.

Dynamic tuning of Master/Worker applications. P. Caymes Scutari, A. Morajko, T. Margalef, E. Cesar, J. Sorribes, E. Luque. Paradyn/Condor Week 2005. 14-18 March 2005. Madison (Wisconsin), United States.

However, that has been a very restrictive configuration, because on the one hand it forced the users to develop their applications using the framework, or on the other hand the users were deeply involved in the complexity of MATE when defining a tunlet, rather than in their applications and the performance problems, when developed the applications using another tools.

In this work, we proposed a methodology to specify tunlets by considering the programming model followed by the application and the performance problem. Thus, users have to think in the application and the performance model as MATE does. A set of abstractions have to be defined, such as the actors in the application (i.e. the different kinds of processes co-executing in the application), the events and information to be collected, the performance parameters and the tuning points. For each entity the user has to provide some information such as data type, location, name, etc. These abstractions will be formalized to constitute a specification.

We studied the viability of automating the creation of tunlets and the requirements to achieve such purpose. Then, we designed a Tunlet Specification Language to formalize the abstractions defined by the user, and in addition we developed and coded a translator to transform a given specification into a tunlet. Such developments were made considering the requirements of MATE, i.e. the API the tunlets have to follow to works in MATE. The translator includes the automation to generate the code corresponding to CPs and Global Analyzer. A summary of the methodology and the language can be found in:

Automatic generation of dynamic tuning techniques. P. Caymes-Scutari, A. Morajko, T.Margalef, E.Luque. Euro-Par 2007. 28-31 August 2007. Laboratorio IRISA, Rennes, France. *Accepted.*

Generación Automática de Técnicas de Sintonización Dinámica. P. Caymes-Scutari, A. Morajko, T.Margalef, J. Sorribes,

E.Luque. XVII Jornadas de Paralelismo-Albacete 2006. Proceedings of the XVII Jornadas de Paralelismo, p. 383-388. Universidad de Castilla-La Mancha, ISBN:84-690-0551-0. 18/09/06 - 20/09/06. Albacete, Spain.

In addition, this work was presented in the Paradyn-Condor Week 2007:

Automatic performance tuning: automatic development of tunlets. P. Caymes Scutari, A. Morajko, T.Margalef, E.Luque. Paradyn/Condor Week 2007. University of Wisconsin. Madison (Wisconsin), United States. 30/04/07 - 03/05/07.

In order to validate the usability of such language, in Chapter 5 we provided two examples of usage, to tune the number of workers and/or the load balance in Master/Worker applications. In addition, these two use cases provide MATE with the logic to automatically tune those problems when the application under consideration was developed using the Master/Worker Framework. In other cases, the tunlets specifications can be re-used to define the specification corresponding to another particular application.

The development of this methodology constitutes a very promising means to extend the use of MATE, due to the users can specify different performance problems for different applications. They will not be restricted to tunlets provided *a priori* by MATE nor involved in the implementing details of MATE.

Both scalability and transparency of MATE are qualities necessary to make MATE a more useful and user-friendly tool. The proposals and developments presented in this work attempt to provide MATE with such characteristics. Experiments showed the viability of the proposals. Even though several improvements remain to extend and improve MATE, we established the basis to scale the analysis process and provide the users the possibility of using MATE in the tuning process of their parallel applications.

6.2 Open Lines

During the development of this thesis, we discovered a lot of topics that can be the goal of future work. Some of these open lines have a direct connection

with our work, and others are related to more general topics. In the following paragraphs we highlight some possible lines to continue the work.

As mentioned before, in this work we established a new approach to execute the analysis phase in MATE. The distributed-hierarchical collecting-preprocessing approach provides MATE with scalability properties when the size of the application increases. However, the initial proposal and implementation could be improved in several manners. As mentioned in Chapter 3, in our prototype we decided beforehand the number of CPs according to the necessities of the application. However, in the future this approach should be applied in a more general way, then we have to provide MATE with knowledge to decide how many CPs are needed according to the current conditions of the system. Thus, a performance model to decide the number of CPs should be defined. Such a performance model, should take into consideration the trade off between the benefits obtained and the resources involved in the analysis process. In addition, the hierarchical levels at collecting-preprocessing stage should be defined when the amount of events to manage in the system is such that the Global Analyzer turns in a bottleneck such as in full centralized approach.

With respect to the methodology to specify tunlets, in this work we provided a tunlet specification language and a translator to automatically generate the tunlets. Even though the translator works over XML specifications, we provided the users with a previous stage in which they can define the specification by following the syntax of the language to specify the tunlet. This exempts the users from being worried about XML labels and syntax. In addition, we could add a more abstract stage in order to increase the user-friendliness of the tunlets specifier.

If we consider more general aspects of MATE to be extended or improved, we can mention the co-execution of tunlets. Until now, MATE just supports the execution of a simple tunlet. Thus, only one performance problem can be solved when an application is executing. Even though it would be very useful the overcoming of several performance problems along the same execution, it is not a trivial nor straightforwardly applicable approach due to the decisions made by a tunlet could be opposite or inconsistent with the decisions made by another tunlet. Then, the study of super-performance models and its

automation should be studied in order to increase the power of MATE. At the same time, it would allow the use of the tunlet to auto-tune the number of CPs.

Another aspect to consider is the augment in the variety of programming models which could use MATE. The current implementation of MATE is oriented to Master/Worker applications, but it would be interesting extend its usability to another programming models, such as Pipeline, Divide&Conquer, etc. Furthermore, once MATE is adapted to work with other programming models than Master/Worker, it should be useful associate to MATE some frameworks or skeletons to develop applications following such models, and in such ways provide a set of predefined tunlets over the frameworks, ready to be used, such as we did in this work, providing two tunlets to the Master/Worker Framework.

Finally, a bigger usability of MATE could be reached when all the implementation of MATE is adapted to work with *mpi* applications too. The initial implementation of MATE is defined using *pvm* to take advantages, for example, from *tasker* and *hoster* services. Currently, some parts of MATE are available to *mpi* applications, but other parts are remaining to provide the complete tuning tool.

Parallel computing is as useful as well as complex field. Due to the constant evolution of this computing area, tools have to evolve in consequence in order to help and simplify the users' task and take the major profit possible of capacities offered by parallelism. In particular, it is necessary to tackle the problem of performance tuning from many different ways. We trust our work is an important contribution from automatic and dynamic tuning to overcome some problems. We hope investigation continues on this area, then we will be able to see in an immediate future the results of such advances.

Appendix A

Tunlet Specification Language: Syntax Directed Definition

IN Chapter 4 we defined a language to specify tunlets. The idea of such a language is to make MATE transparent to the users in order to facilitate the inclusion of knowledge about performance problems. For legibility and space reasons, in Section 4.5.5 we just presented the starting production with its associated semantic rule. In this appendix, we present the whole syntax directed definition. Note that we present the semantic rules following the right part of each production, rather than in a table as proposed in [1].

In order to define the semantic rules for every production, we transformed slightly the grammar presented in Section 4.5.4, by elaborating on the closure of some productions (such as Ev or Ac in which we implemented the "+" as LDEV and LDAC, respectively). Thus, several new productions representing lists of entities are added in the grammar (LVBLEREF, LVBLE, LDEV, LDAC, LATTR and LWORD).

In general, the basic non-terminal symbols have only a "v" attribute, which stores the value associated to such symbol[1]. These attributes "v" are used in the upper levels to compose the different attributes of the symbols (such as IDAC.id or IDAC.min). Finally, non-terminal containing some list of x elements as attribute, has an attribute xs and an index i used to indicate the tail of the list. This index is initialized in 0.

[1]For NUMBER and WORD we assume the "v" attribute, to simplify the definition

Even though we provide the whole syntax-directed definition, the most interesting semantic rule is associated with the start symbol $\mathcal{S}$. This semantic rule condenses the operation of the translation from the specification to the tunlet. The remaining rules, in general, allow for obtaining -*synthetizing*- the values of the attributes. The procedures used to translate the specification into a tunlet were explained in Section 4.5.5

$\mathcal{S} \longrightarrow$ `TUNLET` Tunlet
`MEASURE POINTS` Vv Ac Ii Pp
`PERFORMANCE FUNCTIONS` Fc
`TUNING POINTS` Tp
`ENDTUNLET`
{
Translate_ and_ Solve_ Dependences()
Create_ Tunlet_ Stats()
Create_ Tunlet()
}

Tunlet $\longrightarrow$	`id`: WORD	{Tunlet.id:=WORD.v
	`comment`: LWORD_1	Tunlet.comment:=LWORD_1.words
	`include`: LWORD_2	Tunlet.include:=LWORD_2.words}

Vv $\longrightarrow$ LVBLE
{Vv.vbles:=LVBLE.vbles}

Ev $\longrightarrow$ `EVENTS` LDEV
{Ev.events:=LDEV.events}

Ac $\longrightarrow$ `ACTORS` LDAC
{Ac.actors:=LDAC.actors}

li ⟶ `ITERATION INFORMATION` LATTR
{li.attrs:=LATTR.attrs}

Pp ⟶ `MODEL PARAMETERS` LATTR
{Pp.attrs:=LATTR.attrs}

Fc ⟶ Fc_1 DELIMF
{Fc.funcs[Fc.i].def:=DELIMF.def
Fc.i++}

Fc ⟶ DELIMF
{Fc.funcs[Fc.i].def:=DELIMF.def
Fc.i++}

Tp ⟶ Tp_1 DELIMP
{Tp.points[Tp.i].id:=DELIMP.id
Tp.points[Tp.i].value:=DELIMP.value
Tp.points[Tp.i].kind:=DELIMP.kind
Tp.points[Tp.i].syncfunction:=DELIMP.syncfunction
Tp.points[Tp.i].syncplace:=DELIMP.syncplace
Tp.points[Tp.i].cond:=DELIMP.cond
Tp.points[Tp.i].place:=DELIMP.place
Tp.points[Tp.i].idx:=DELIMP.idx
Tp.points[Tp.i].req:=DELIMP.req
Tp.points[Tp.i].attrs:=DELIMP.attrs
Tp.i++}

Tp ⟶ DELIMP
{Tp.points[Tp.i].id:=DELIMP.id
Tp.points[Tp.i].value:=DELIMP.value
Tp.points[Tp.i].kind:=DELIMP.kind
Tp.points[Tp.i].syncfunction:=DELIMP.syncfunction
Tp.points[Tp.i].syncplace:=DELIMP.syncplace
Tp.points[Tp.i].cond:=DELIMP.cond
Tp.points[Tp.i].place:=DELIMP.place
Tp.points[Tp.i].idx:=DELIMP.idx
Tp.points[Tp.i].req:=DELIMP.req
Tp.points[Tp.i].attrs:=DELIMP.attrs
Tp.i++}

LVBLEREF ⟶ LVBLEREF_1 VBLEREF
{LVBLEREF.vblesr[LVBLEREF.i].id:=VBLEREF.id
LVBLEREF.i++}
LVBLEREF ⟶ VBLEREF
{LVBLEREF.vblesr[LVBLEREF.i].id:=VBLEREF.id
LVBLEREF.i++}

LVBLE ⟶ LVBLE_1 DELIMV
{ LVBLE.vbles[LVBLE.i].id:=DELIMV.id
LVBLE.vbles[LVBLE.i].comment:=DELIMV.comment
LVBLE.vbles[LVBLE.i].source:=DELIMV.source
LVBLE.vbles[LVBLE.i].type:=DELIMV.type
LVBLE.vbles[LVBLE.i].actorId:=DELIMV.actorId
LVBLE.i++ }

LVBLE ⟶ DELIMV
{ LVBLE.vbles[LVBLE.i].id:=DELIMV.id
LVBLE.vbles[LVBLE.i].comment:=DELIMV.comment
LVBLE.vbles[LVBLE.i].source:=DELIMV.source
LVBLE.vbles[LVBLE.i].type:=DELIMV.type
LVBLE.vbles[LVBLE.i].actorId:=DELIMV.actorId
LVBLE.i++ }

LDEV ⟶ LDEV_1 DELIMEV
{LDEV.events[LDEV.i].id:=DELIMEV.id
LDEV.events[LDEV.i].actorId:=DELIMEV.actorId
LDEV.events[LDEV.i].controliter:=DELIMEV.controliter
LDEV.events[LDEV.i].utility:=DELIMEV.utility
LDEV.events[LDEV.i].method:=DELIMEV.method
LDEV.events[LDEV.i].class:=DELIMEV.class
LDEV.events[LDEV.i].place:=DELIMEV.place
LDEV.events[LDEV.i].attrs:=DELIMEV.attrs
LDEV.i++}

LDEV ⟶ DELIMEV
{LDEV.events[LDEV.i].id:=DELIMEV.id
LDEV.events[LDEV.i].actorId:=DELIMEV.actorId
LDEV.events[LDEV.i].controliter:=DELIMEV.controliter
LDEV.events[LDEV.i].utility:=DELIMEV.utility
LDEV.events[LDEV.i].method:=DELIMEV.method
LDEV.events[LDEV.i].class:=DELIMEV.class
LDEV.events[LDEV.i].place:=DELIMEV.place
LDEV.events[LDEV.i].attrs:=DELIMEV.attrs
LDEV.i++}

LDAC ⟶ LDAC_1 DELIMAC
{LDAC.actors[LDAC.i].id:=DELIMAC.id
LDAC.actors[LDAC.i].min:=DELIMAC.min
LDAC.actors[LDAC.i].max:=DELIMAC.max
LDAC.actors[LDAC.i].completion:=DELIMAC.completion
LDAC.actors[LDAC.i].class:=DELIMAC.class
LDAC.actors[LDAC.i].exe:=DELIMAC.exe
LDAC.actors[LDAC.i].attrs:=DELIMAC.attrs
LDAC.i++}

LDAC ⟶ DELIMAC
{LDAC.actors[LDAC.i].id:=DELIMAC.id
LDAC.actors[LDAC.i].min:=DELIMAC.min
LDAC.actors[LDAC.i].max:=DELIMAC.max
LDAC.actors[LDAC.i].completion:=DELIMAC.completion
LDAC.actors[LDAC.i].class:=DELIMAC.class
LDAC.actors[LDAC.i].exe:=DELIMAC.exe
LDAC.actors[LDAC.i].attrs:=DELIMAC.attrs
LDAC.i++}

LATTR ⟶ LATTR_1 ATTR
{LATTR.attrs[LATTR.i].id:=ATTR.id
LATTR.attrs[LATTR.i].comment:=ATTR.comment
LATTR.attrs[LATTR.i].type:=ATTR.type
LATTR.attrs[LATTR.i].inic:=ATTR.inic
LATTR.attrs[LATTR.i].depinic:=ATTR.depinic
LATTR.attrs[LATTR.i].value:=ATTR.value
LATTR.attrs[LATTR.i].cum:=ATTR.cum
LATTR.attrs[LATTR.i].depdcy:=ATTR.depdcy
LATTR.i++}

LATTR ⟶ ATTR
{LATTR.attrs[LATTR.i].id:=ATTR.id
LATTR.attrs[LATTR.i].comment:=ATTR.comment
LATTR.attrs[LATTR.i].type:=ATTR.type
LATTR.attrs[LATTR.i].inic:=ATTR.inic
LATTR.attrs[LATTR.i].depinic:=ATTR.depinic
LATTR.attrs[LATTR.i].value:=ATTR.value
LATTR.attrs[LATTR.i].cum:=ATTR.cum
LATTR.attrs[LATTR.i].depdcy:=ATTR.depdcy
LATTR.i++}

LWORD ⟶ LWORD_1 WORD
{LWORD.words[LWORD.i].id:=WORD.v
LWORD.i++}

LWORD ⟶ WORD
{LWORD.words[LWORD.i].id:=WORD.v
LWORD.i++}

DELIMV ⟶ `variable` VBLE `endvariable`
{ DELIMV.id :=VBLE.id
DELIMV.comment :=VBLE.comment
DELIMV.source:=VBLE.source
DELIMV.type:=VBLE.type
DELIMV.actorId:=VBLE.actorId }

VBLE ⟶	`id:` WORD_1	{ VBLE.id:=WORD_1.v
	`comment:` LWORD	VBLE.comment:=LWORD.words
	`source:` SOURCE	VBLE.source:=SOURCE.v
	`type:` TYPE	VBLE.type:=TYPE.v
	`actorId:` WORD_2	VBLE.type:=WORD_2.v }

DELIMEV ⟶ `event` EVENT `endevent`
{ DELIMEV.id :=EVENT.id
DELIMEV.actorId :=EVENT.actorId
DELIMEV.method:=EVENT.method
DELIMEV.class:=EVENT.class
DELIMEV.controliter:=EVENT.controliter
DELIMEV.utility:=EVENT.utility
DELIMEV.place:=EVENT.place
DELIMEV.attrs:=EVENT.attrs}

EVENT ⟶ IDEV (`ATTRS` LVBLEREF)?
{ EVENT.id :=IDEV.id
EVENT.actorId :=IDEV.actorId
EVENT.method:=IDEV.method
EVENT.class:=IDEV.class
EVENT.controliter:=IDEV.controliter
EVENT.utility:=IDEV.utility
EVENT.place:=IDEV.place
EVENT.attrs:=LVBLEREF.vblesr}

IDEV ⟶	`id:` WORD_1	{ IDEV.id:=WORD_1.v
	`actor:` WORD_2	IDEV.actorId:=WORD_2.v
	`method:` WORD_3	IDEV.method:=WORD_3.v
	`class:` WORD_4	IDEV.class:=WORD_4.v
	`controliter:` CODE	IDEV.controliter:=CODE.v
	`utility:` UTIL	IDEV.utility:=UTIL.v
	`place:` INSTPL	IDEV.place:=INSTPL.v }

DELIMAC ⟶ `actor` ACTOR `endactor`
{DELIMAC.id:=ACTOR.id
DELIMAC.min:=ACTOR.min
DELIMAC.max:=ACTOR.max
DELIMAC.completion:=ACTOR.completion
DELIMAC.class:=ACTOR.class
DELIMAC.exe:=ACTOR.exe
DELIMAC.attrs:=ACTOR.attrs }

ACTOR ⟶ IDAC (`ATTRS` (LATTR)$^+$)?
{ACTOR.id:=IDAC.id
ACTOR.min:=IDAC.min
ACTOR.max:=IDAC.max
ACTOR.completion:=IDAC.completion
ACTOR.class:=IDAC.class
ACTOR.exe:=IDAC.exe
ACTOR.attrs:=LATTR.attrs }

IDAC ⟶	`id:` WORD_1	{ IDAC.id:=WORD_1.v
	`min:` NUMBER_1	IDAC.min:=NUMBER_1.v
	`max:` NUMBER_2	IDAC.max:=NUMBER_2.v
	`completion:` EXP	IDAC.completion:=EXP.v
	`class:` WORD_2	IDAC.class:=WORD_2.v
	`exe:` WORD_3	IDAC.exe:=WORD_3.v }

DELIMF ⟶ `func def:` EXP `endfunc`
{DELIMF.def:=EXP.v }

DELIMP ⟶ `point`
VBLEREF POINT (`ATTRS` LVBLEREF)?
`endpoint`
{DELIMP.id:=VBLEREF.id
DELIMP.value:=POINT.value
DELIMP.kind:=POINT.kind
DELIMP.syncfunction:=POINT.syncfunction
DELIMP.syncplace:=POINT.syncplace
DELIMP.cond:=POINT.cond
DELIMP.place:=POINT.place
DELIMP.idx:=POINT.idx
DELIMP.req:=POINT.req
DELIMP.attrs:=LISVBLEREF.vblesr }

POINT ⟶	`value:` EXP_1	{POINT.value:=EXP_1.v
	`kind:` KIND	POINT.kind:=KIND.v
	`syncfunction:` WORD	POINT.syncfunction:=WORD.v
	`syncplace:` INSTPL	POINT.syncplace:=INSTPL.v
	`cond:` EXP_2	POINT.cond:=EXP_2.v
	(`idx:` NUMBER	(POINT.idx:=NUMBER.v
	`req:` LOG) ?	POINT.req:=LOG.v) ?
	(`place:` INSTPL) ?	(POINT.place:=INSTPL.v) ? }

ATTR ⟶	`id:` $WORD_1$	{ ATTR.id:=$WORD_1$.v
	`comment:` LWORD	ATTR.comment:=LWORD.words
	`type:` TYPE	ATTR.type:=TYPE.v
	`inic:` EXP_1	ATTR.inic:=EXP_1.v
	`depinic:` VAL_1	ATTR.depinic:=VAL_1.v
	`value:` EXP_2	ATTR.value:=EXP_2.v
	`cum:` LOG	ATTR.cum:=LOG.v
	`dependency:` VAL_2	ATTR.depdcy:=VAL_2.v }

VBLEREF ⟶ `id`:WORD { VBLEREF.id:=WORD.v}

VAL ⟶ WORD { VAL.v:=WORD.v}
VAL ⟶ `none` { VAL.v:="none"}

NUMBER ⟶ [`0..9`]$^+$

WORD ⟶ ([`a..z`]||[`A..Z`])([`0..9`]||[`a..z`]||[`A..Z`]||
. | _ | (|) | : | / | * | [|] | ;)*

EXP ⟶ C/C++FUNCTION
{EXP.v:=C/C++FUNCTION.v}
EXP ⟶ C/C++EXP
{EXP.v:=C/C++EXP.v}

INSTPL ⟶ `entry` {INSTPL.v:="*entry*"}
INSTPL ⟶ `exit` {INSTPL.v:="*exit*"}

TYPE ⟶ `int` {TYPE.v:="*int*"}
TYPE ⟶ `short` {TYPE.v:="*short*"}
TYPE ⟶ `float` {TYPE.v:="*float*"}
TYPE ⟶ `double` {TYPE.v:="*double*"}
TYPE ⟶ `char` {TYPE.v:="*char*"}
TYPE ⟶ `string` {TYPE.v:="*string*"}

SOURCE ⟶ `asFuncParamValue`
{SOURCE.v:="*asFuncParamValue*"}
SOURCE ⟶ `asVarValue`
{SOURCE.v:="*asVarValue*"}
SOURCE ⟶ `asFuncReturnValue`
{SOURCE.v:="*asFuncReturnValue*"}

SOURCE ⟶ `asConstValue`
{SOURCE.v:=“*asConstValue*”}
SOURCE ⟶ `asFuncParamPointerValue`
{SOURCE.v:=“*asFuncParamPointerValue*”}

CODE ⟶ `begin`
{CODE.v:=“*begin*”}
CODE ⟶ `end`
{CODE.v:=“*end*”}
CODE ⟶ `eval`
{CODE.v:=“*eval*”}
CODE ⟶ `no`
{CODE.v:=“*no*”}

UTIL ⟶ `always`
{UTIL.v:=“*always*”}
UTIL ⟶ `addable`
{UTIL.v:=“*addable*”}
UTIL ⟶ `eval`
{UTIL.v:=“*eval*”}

LOG ⟶ `true`
{LOG.v:=“*true*”}
LOG ⟶ `false`
{LOG.v:=“*false*”}

KIND ⟶ `SetVariableValue`
{KIND.v:=“*SetVariableValue*”}
KIND ⟶ `ReplaceFunction`
{KIND.v:=“*ReplaceFunction*”}

KIND ⟶ `InsertFunctionCall`
{KIND.v:=“*InsertFunctionCall*”}
KIND ⟶ `RemoveFuncCall`
{KIND.v:=“*RemoveFuncCall*”}
KIND ⟶ `OnTimeFuncCall`
{KIND.v:=“*OnTimeFuncCall*”}
KIND ⟶ `FuncParamChange`
{KIND.v:=“*FuncParamChange*”}

Appendix B

Tunlet Specification: Optimal Number of Workers

IN this appendix we present the complete specification of the tunlet to tune the number of workers. In Section 5.1 we documented the followed steps in order to obtain this specification.

```
TUNLET
name:nworkers
comment:/*tunlet to tune the number of workers
          in M/W applications. If the applications
          are developed using the Framework
          Master/Worker associated to MATE, the
          tunlet can be straightfordwardly used,
          i.e. any changes or modifications are
          required in this specification.*/
include: math.h

MEASURE POINTS
   VARIABLES AND VALUES

     variable
       id: iteration
       comment: /*indicates the current iteration
```

```
            of the master process*/
  source: asVarValue
  type: int
  actorId:master
endvariable

variable
  id: TheWorkUnitBytes
  comment:/*amount of bytes sent in a task
           by the master to each worker*/
  source: asVarValue
  type: int
  actorId:master
endvariable

variable
  id: workerTID
  comment:/*worker TID*/
  source: asVarValue
  type: int
  actorId:master
endvariable

variable
  id: numtuples
  comment:/*amount of tasks sent to a worker*/
  source: asVarValue
  type: int
  actorId:master
endvariable

variable
  id: workerTIDr
  comment:/*answering worker TID*/
```

```
  source: asVarValue
  type: int
  actorId:master
endvariable

variable
  id: nbytes
  comment:/*amount of bytes of the answer*/
  source: asVarValue
  type: int
  actorId:master
endvariable

variable
  id:CurrentIteration
  comment:/*current iteration of the worker*/
  source: asVarValue
  type: int
  actorId:worker
endvariable

variable
  id: nw
  comment: /*current number of workers*/
  source: asVarValue
  type: int
  actorId:master
endvariable

variable
  id: NOptWorkers
  comment:/*optimal number of workers according
            to the current conditions of the system */
  source: asVarValue
```

```
    type: int
    actorId:master
  endvariable

EVENTS

  event
    id:IterationStarts
    actorId:master
    controliter:begin
    utility:always
    method: _M_SendIteration
    class:_CMaster
    place: entry
  ATTRS
    id: iteration
    id: TheWorkUnitBytes
    id: TheNumTuples
    id: nw
  endevent

  event
    id: IterationFinishes
    actorId: master
    controliter:end
    utility:always
    method:_M_ReceiveIteration
    class: _CMaster
    place: exit
  ATTRS
    id: iteration
  endevent

  event
```

```
  id:MSendsTaskW
  actorId:master
  controliter:no
  utility:always
  method:_DMM_SendTask
  class: _CDtaMngM
  place: exit
ATTRS
  id:workerTID
  id:numtuples
endevent

event
  id:WRepliesM
  actorId:master
  controliter:no
  utility:always
  method:_DMM_ReceiveEffective
  class: _CDtaMngM
  place:exit
ATTRS
  id:workerTIDr
  id:nbytes
endevent

event
  id:WStartsTask
  actorId:worker
  controliter:no
  utility:always
  method:_W_DoWorks
  class: _CWorker
  place: entry
ATTRS
```

```
    id:CurrentIteration
  endevent

  event
    id:WFinishesTask
    actorId:worker
    controliter:eval
    utility:always
    method:_W_SendAll
    class: _CWorker
    place:entry
  ATTRS
    id:CurrentIteration
  endevent

ACTORS
  actor
    id: master
    min: 1
    max: 1
    completion:/#comp==1#/
    class:_CMaster,  _MyMaster
    exe: /home/paola/pvm3/bin/LINUX/master
  ATTRS
    id:comp
    comment:/*when an iteration finishes, the value of comp
              is set to 1, which indicates the completion of
              the master.*/
    type: int
    inic:/#comp=0;#/
    depinic:none
    value: /#master[0].comp=1;#/
    cum:false
    dependency: IterationFinishes
```

```
id:firstSend
comment:/*instant in which the master executes the first
          send of the iteration*/
type: double
inic:/#firstSend=0.0;#/
depinic:none
value: /#if(MSendsTaskW.timestamp<master[0].firstSend ||
            master[0].firstSend==0.0)
          master[0].firstSend= MSendsTaskW.timestamp;#/
cum:false
dependency: MSendsTaskW

id:lastRecv
comment:/*instant in which the master receives the last
          answer of the iteration*/
type: double
inic:/#lastRecv=0.0;#/
depinic:none
value:/# if(WRepliesM.timestamp>master[0].lastRecv ||
            master[0].lastRecv==0.0 )
           master[0].lastRecv= WRepliesM.timestamp;#/
cum:false
dependency: WRepliesM

id:lastWorker
comment:/*id of the worker which answered at last.
          It is needed to obtain its computing time.*/
type: int
inic:/#lastWorker=0;#/
depinic:none
value: /#if(WRepliesM.timestamp>master[0].lastRecv ||
            master[0].lastRecv==0.0)
          master[0].lastWorker= WRepliesM.id;#/
```

```
  cum:false
  dependency: WRepliesM
endactor

actor
  id: worker
  min:1
  max: 18
  completion:/#comp==1#/
  class: _CWorker, _MyWorker
  exe: /home/paola/pvm3/bin/LINUX/worker
ATTRS

  id:comp
  comment:/*when the processing of the task finishes, the
            value of comp is set to 1, which indicates the
            completion of the worker.*/
  type: int
  inic:/#comp=0;#/
  depinic:none
  value: /#worker[WFinishesTask.id].comp=1;#/
  cum:false
  dependency: WFinishesTask

  id:timestampRecv
  comment:/*instant in which the worker receive the data*/
  type: double
  inic: /#timestampRecv=0.0;#/
  depinic:none
  value:/#worker[MSendsTaskW.workerTID].timestampRecv=
                    MSendsTaskW.timestamp;#/
  cum:false
  dependency:MSendsTaskW
```

```
id: timestampSnd
comment:/*instant in which the master receives an answer
          from the worker workerTIDr*/
type: double
inic: /#timestampSnd=0.0;#/
depinic:none
value:/#worker[WRepliesM.workerTIDr].timestampSnd=
                   WRepliesM.timestamp;#/
cum:false
dependency:WRepliesM

id: numtuplesRecv
comment: /*amount of tuples to be processed by the worker*/
type: int
inic:/# numtuplesRecv=0;#/
depinic:none
value:/# worker[MSendsTaskW.workerTID].numtuplesRecv=
                   MSendsTaskW.numtuples;#/
cum:false
dependency:MSendsTaskW

id:replysize
comment: /*size of the results sent by the worker*/
type: int
inic: /#replysize=0;#/
depinic:none
value: /#worker[WRepliesM.workerTIDr].replysize=
                   WRepliesM.nbytes;#/
cum:false
dependency:WRepliesM

id:timestampStartCalc
comment:/*instant in which the worker starts processing
          the data*/
```

```
    type: double
    inic:/#timestampStartCalc=0.0;#/
    depinic:none
    value:/#worker[WStartsTask.id].timestampStartCalc=
                        WStartsTask.timestamp;#/
    cum:false
    dependency:WStartsTask

    id:timestampEndCalc
    comment:/*instant in which the worker finishes processing
              the data*/
    type: double
    inic:/#timestampEndCalc=0.0;#/
    depinic:none
    value:/#worker[WFinishesTask.id].timestampEndCalc=
                        WFinishesTask.timestamp;#/
    cum:false
    dependency:WFinishesTask

    id:cti
    comment:/*computing time of the worker*/
    type: double
    inic: /#cti=0.0;#/
    depinic:none
    value: /#worker[WFinishesTask.id].cti=
               worker[WFinishesTask.id].timestampEndCalc -
               worker[WFinishesTask.id].timestampStartCalc;#/
    cum:false
    dependency: timestampEndCalc
  endactor

ITERATION INFORMATION
  id: StartedIteration
  comment: /*indicates the number of the last
```

```
            (or current)iteration*/
  type: int
  inic: /#StartedIteration=0;#/
  depinic:none
  value: /#iter.StartedIteration=
              IterationStarts.iteration;#/
  cum:false
  dependency:IterationStarts

  id: tuplesize
  comment: /*indicates the size of each task,
            in bytes*/
  type: int
  inic: /#tuplesize=0;#/
  depinic:none
  value: /#iter.tuplesize=
           IterationStarts.TheWorkUnitBytes;#/
  cum:false
  dependency:IterationStarts

  id: iterCommTime
  comment: /*communication time of the iteration*/
  type: double
  inic: /#iterCommTime=0.0;#/
  depinic:none
  value: /#iter.iterCommTime=
            (master[0].lastRecv - master[0].firstSend)-
              (worker[master[0].lastWorker].cti);#/
  cum:false
  dependency:IterationFinishes


MODEL PARAMETERS
  id:n
```

```
comment: /*current number of workers*/
type:int
inic: /#n=0;#/
depinic:none
value: /#n=iter.nw;#/
cum:false
dependency:none

id:tl
comment: /*latency - constant*/
type:float
inic: /#tl=1000;#/
depinic:none
value: /#tl=1000;#/
cum:false
dependency:none

id:vi
comment: /*total data volume to be processed*/
type:int
inic: /#vi=0;#/
depinic:none
value: /#vi=0;for(int i=0;i<n;i++)vi+=
            worker[i].numtuplesRecv*iter.tuplesize;#/
cum:true
dependency:none

id:viuno
comment: /*data volume sent to a worker*/
type:int
inic: /#viuno=0;#/
depinic:none
value: /#viuno=vi/n;#/
cum:false
```

```
dependency:vi

id:vm
comment: /*volume of answers*/
type:int
inic: /#vm=0;#/
depinic:none
value: /#vm=0;for(int i=0;i<n;i++)vm+=worker[i].replysize;#/
cum:true
dependency:none

id:vmuno
comment: /*volume of an answer*/
type:int
inic: /#vmuno=0;#/
depinic:none
value: /#vmuno=vm/n;#/
cum:false
dependency:vm

id:lambda
comment: /*cost of sending a byte*/
type:float
inic: /#lambda=0.0;#/
depinic:none
value: /#lambda=iter.iterCommTime/(vi+vmuno);#/
cum:false
dependency:vmuno

id:Ct
comment: /*total computing time*/
type:double
inic: /#Ct=0.0;#/
depinic:none
```

```
     value: /#Ct=0;for(int i=0;i<n;i++)Ct+=worker[i].cti;#/
     cum:true
     dependency:none

     id:Vt
     comment: /*total managed volume of data */
     type:int
     inic: /#Vt=0;#/
     depinic:none
     value: /#Vt=vm+vi;#/
     cum:false
     dependency:vm


PERFORMANCE FUNCTION

     function
        def:/#int pf(){ int nro=0; nro=
             (int)sqrt( (lambda * Vt +Ct)/tl);return nro;}#/
     endfunction


TUNING POINTS
     point
       id:NOptWorkers
       value:/#pf()#/
       kind: SetVariableValue
       syncfunction:0
       syncplace:0
       cond:/#NOptWorkers>iter.GetNum_worker()+2 ||
             NOptWorkers<iter.GetNum_worker()-2 #/
     endpoint
ENDTUNLET
```

Appendix C

Tunlet Specification: Load Balancing

IN this appendix we present the complete specification of the tunlet to tune the load balancing. In Section 5.2 we documented the followed steps in order to obtain this specification.

```
TUNLET
name:nworkers
comment:/*tunlet to tune the load balancing
          in M/W applications. If the applications
          are developed using the Framework
          Master/Worker associated to MATE, the
          tunlet can be straightfordwardly used,
          i.e. any changes or modifications are
          required in this specification.*/
include:math.h

MEASURE POINTS
   VARIABLES AND VALUES

     variable
       id: iteration
       comment: /*indicates the current iteration
```

```
            of the master process*/
  source: asVarValue
  type: int
  actorId:master
endvariable

variable
  id: TheNumTuples
  comment:/*total amount of tasks of
           the iteration*/
  source: asVarValue
  type: int
  actorId:master
endvariable

variable
  id: TheWorkUnitBytes
  comment:/*size of a task*/
  source: asVarValue
  type: int
  actorId:master
endvariable

variable
  id: workerTID
  comment:/*worker TID*/
  source: asVarValue
  type: int
  actorId:master
endvariable

variable
  id: numtuples
  comment:/*amount of tasks sent to a worker*/
```

```
  source: asVarValue
  type: int
  actorId:master
endvariable

variable
  id: workerTIDr
  comment:/*answering worker TID*/
  source: asVarValue
  type: int
  actorId:master
endvariable

variable
  id:CurrentIteration
  comment:/*current iteration of the worker*/
  source: asVarValue
  type: int
  actorId:worker
endvariable

variable
  id: NroTasks
  comment: /*number of task to proces by worker*/
  source: asVarValue
  type: int
  actorId:worker
endvariable

variable
  id: nw
  comment: /*current number of workers*/
  source: asVarValue
  type: int
```

```
    actorId:master
  endvariable

  variable
    id: globalSizeChunk
    comment:/*size of the chunk,
              is the tunable variable*/
    source: asVarValue
    type: int
    actorId:master
  endvariable

EVENTS
  event
    id:IterationStarts
    actorId:master
    controliter:begin
    utility:always
    method: _M_SendIteration
    class:_CMaster
    place: entry
  ATTRS
    id: iteration
    id: TheNumTuples
    id: TheWorkUnitBytes
    id: nw
  endevent

  event
    id: IterationFinishes
    actorId: master
    controliter:end
    utility:always
    method:_M_ReceiveIteration
```

```
  class: _CMaster
  place: exit
ATTRS
  id:iteration
endevent

event
  id:MSendsTaskW
  actorId:master
  controliter:no
  utility:always
  method:_DMM_SendTask
  class: _CDtaMngM
  place: exit
ATTRS
  id:workerTID
  id:numtuples
endevent

event
  id:WRepliesM
  actorId:master
  controliter:no
  utility:always
  method:_DMM_ReceiveEffective
  class: _CDtaMngM
  place:exit
ATTRS
  id:workerTIDr
endevent

event
  id:WStartsTask
  actorId:worker
```

```
      controliter:no
      utility:always
      method:_W_DoWorks
      class: _CWorker
      place: entry
    ATTRS
      id:CurrentIteration
      id:NroTasks
    endevent

    event
      id:WFinishesTask
      actorId:worker
      controliter:eval
      utility:always
      method:_W_SendAll
      class: _CWorker
      place:entry
    ATTRS
      id:CurrentIteration
    endevent

ACTORS
  actor
    id: master
    min: 1
    max: 1
    completion:/#comp==1#/
    class:_CMaster,  _MyMaster
    exe: /home/paola/pvm3/bin/LINUX/master
  ATTRS
    id:comp
     comment:/*when an iteration finishes, the value of comp
               is set to 1, which indicates the completion of
```

```
            the master.*/
   type: int
   inic:/#comp=0;#/
   depinic:none
   value: /#master[0].comp=1;#/
   cum:false
   dependency: IterationFinishes
endactor

actor
  id: worker
  min:1
  max: 25
  completion:/#comp==nrobatch#/
  class: _CWorker, _MyWorker
  exe: /home/paola/pvm3/bin/LINUX/worker
ATTRS
  id: comp
  comment: /*to register the completion*/
  type: int
  inic:/# comp=0;#/
  depinic:none
  value:/#worker[WFinishesTask.id].comp=
           worker[WFinishesTask.id].comp + 1;#/
  cum:false
  dependency:WFinishesTask

  id: numTasksRecv
  comment: /*amount of received tasks*/
  type: int
  inic:/# numTasksRecv=0;#/
  depinic:none
  value:/# worker[WStartsTask.id].numTasksRecv=
     WStartsTask.NroTasks;#/
```

```
cum:true
dependency:WStartsTask

id:timestampStartCalc
comment:/* computing start instant of the worker*/
type: double
inic:/#timestampStartCalc=0.0;#/
depinic:none
value:/#worker[WStartsTask.id].timestampStartCalc=
   WStartsTask.timestamp;#/
cum:false
dependency:WStartsTask

id:timestampEndCalc
comment:/* computing end instant of the worker*/
type: double
inic:/#timestampEndCalc=0.0;#/
depinic:none
value:/#worker[WFinishesTask.id].timestampEndCalc=
                 WFinishesTask.timestamp;#/
cum:false
dependency:WFinishesTask

id:cti
comment:/* computing time worker i*/
type: double
inic: /#cti=0.0;#/
depinic:none
value: /#worker[WFinishesTask.id].cti=
          worker[WFinishesTask.id].cti +
   (worker[WFinishesTask.id].timestampEndCalc
   - worker[WFinishesTask.id].timestampStartCalc);#/
cum:true
dependency: timestampEndCalc
```

```
    id:C
    comment:/* average task computing time of worker i*/
    type: double
    inic: /#C=0.0;#/
    depinic:none
    value: /#worker.C=worker.cti/worker.numTaskRecv;#/
    cum:true
    dependency: cti
  endactor

ITERATION INFORMATION
  id: StartedIteration
  comment: /*current iteration*/
  type: int
  inic: /#StartedIteration=0;#/
  depinic:none
  value: /#iter.StartedIteration=IterationStarts.iteration;
      iter.currentBatch=0; #/
  cum:false
  dependency:IterationStarts

  id: totalworktodo
  comment:/*amount of tasks to be processed*/
  type: int
  inic: /#totalworktodo=0;#/
  depinic:none
  value: /#iter.totalworktodo=IterationStarts.TheNumTuples;#/
  cum:false
  dependency:IterationStarts

  id: tuplesize
  comment: /*task size (in bytes)*/
  type: int
```

```
inic: /#tuplesize=0;#/
depinic:none
value: /#iter.tuplesize=
     IterationStarts.TheWorkUnitBytes;#/
cum:false
dependency:IterationStarts

id: numW
comment: /*amount of workers in the iteration*/
type: int
inic: /#numW=0;#/
depinic:none
value: /#iter.numW=IterationStarts.nw;#/
cum:false
dependency:IterationStarts

id: remainingTasks
comment: /*remaining tasks of the iteration*/
type: int
inic: /#remainingTasks=iter.totalworktodo;#/
depinic:iter.totalworktodo
value: /#remainingTasks=remainingTasks-
                MSendsTaskW.numtuples#/
cum:false
dependency:MSendsTaskW

id: currentBatch
comment: /*current batch */
type: int
inic: /#currentBatch=0#/
depinic:none
value: /##/
cum:false
dependency:none
```

```
  id: remainingTasksBatch
  comment: /*remaing tasks of the batch*/
  type: int
  inic: /#remainingTasksBatch=f[iter.currentBatch]*P;#/
  depinic:iter.currentBatch
  value: /#remainingTasksBatch=remainingTasksBatch
       -MSendsTaskW.numtuples;
       if(remainingTasksBatch==0 &&
          iter.currentBatch<nrobatch-1 )
       {changeFactor();}#/
  cum:false
  dependency:MSendsTaskW

MODEL PARAMETERS
  id:P
  comment: /*current number of workers*/
  type:int
  inic: /#N=0;#/
  depinic:none
  value: /#N=iter.numW;#/
  cum:false
  dependency:none

  id:N
  comment: /*number of task of the iteration*/
  type:int
  inic: /#N=0;#/
  depinic:none
  value: /#N=0;
           for(int i=0; i<P; i++)
           { N=iter.totalworktodo *
               iter.tuplesize); }   #/
  cum:true
```

```
dependency:none

id:R
comment: /*Remaning tasks in the interation*/
type:int
inic: /#R=N;#/
depinic:none
value: /##/
cum:true
dependency:N

id:Ct
comment: /*total computing time*/
type:double
inic: /#Ct=0;#/
depinic:none
value: /#Ct=0;for(int i=0;i<n;i++)Ct+=worker[i].C;#/
cum:true
dependency:none

id:MC
comment: /*mean of C*/
type:double
inic: /#MC=0;#/
depinic:none
value: /#MC=Ct/N;#/
cum:false
dependency:Ct

id:sumMC
comment: /*intermediate parameter to calculate SC*/
type:double
inic: /#sumMC=0.0;#/
depinic:none
```

```
value: /#for(int i=0;i<P;i++){sumMC=sumMC+
         (pow(worker[i].C,2) - pow(MC,2) ) }#/
cum:true
dependency:MC

id:SC
comment: /*standard deviation of C*/
type:double
inic: /#SC=0.0#/
depinic:none
value: /#SC=sqrt(1/N*sumMC);#/
cum:false
dependency:sumMC

id:nrobatch
comment: /*current number of batch*/
type:int
inic: /#nrobatch=0;#/
depinic:none
value: /##/
cum:false
dependency:none

id:x[10]
comment: /*factors to be used through the batchs*/
type:int
inic: /#for(int i=0;i<10;i++){x[i]=0.0;}#/
depinic:none
value: /#for(int i=0;i<10;i++)
          {if(i==0)
           {x[i]=(MC+(SC*sqrt(P/2)))/MC;}
           else
           {x[i]=(2*MC+(SC*sqrt(P/2)))/MC;}
          }#/
```

```
    cum:false
    dependency:none


    id:f[10]
    comment: /*factors to be used through the batchs*/
    type:int
    inic: /#for(int i=0;i<10;i++){f[i]=0;}#/
    depinic:none
    value: /#nrobatch=0;
              while((nrobatch<10) && (R>P*2))
              {if(nrobatch==0)
               {f[nrobatch]=(int)(N/x[nrobatch]*P);}
               else{{f[nrobatch]=(int)(R/x[nrobatch]*P;}
               R=R-(f[nrobatch]*P); nrobatch++;}
               iter.currentBatch=0;#/
    cum:false
    dependency:x



PERFORMANCE FUNCTIONS

     function
        def:/#int pf()
              {int a; a=f[iter.currentBatch+1];
              iter.currentBatch++; return (a);} #/
     endfunction

     function
        def:/#int changefactor()
              { if(f[iter.currentBatch+1] ==
                      f[iter.currentBatch] )
                {Tune_globalSizeChunk();} }#/
```

```
     endfunction

TUNING POINTS
     point
       id:globalSizeChunk
       value:/#pf()#/
       kind: SetVariableValue
       syncfunction:0
       syncplace:0
       cond:/#true#/
     endpoint
ENDTUNLET
```

Glossary

AC : Application Controller. This is the module of MATE which controls the execution of each task of the application. Is composed of several modules such as Monitor and Tuner.

Analyzer : is the module of MATE responsible for evaluating the behaviour of the application. In the distributed-hierarchical collecting-preprocessing approach, the analysis is done in cooperation among CPs and a Global Analyzer.

CP : Collector Preprocessor. In the distributed-hierarchical collecting-preprocessing approach, the CP is each one of the modules responsible for collecting the events incoming from a determined set of machines. In addition, the information carried by the events is classified and preprocessed as much as possible, before sending the condensed relevant information to the Global Analyzer.

DMLib : Dynamic Monitoring Library. It is a shared library used by AC in order to facilitate the instrumentation and data collection. In addition, it is responsible for the registration of events.

DTAPI : is the interface used by Analyzer to represent the application, the tasks, the events and the tuning actions, and to handle the performance monitoring and tuning of the application

DynInst : is a library which implements dynamic instrumentation. It is used by MATE to insert monitoring instrumentation and tuning changes.

Global Analyzer : In the distributed-hierarchical collecting-preprocessing approach, the Global Analyzer is the process which manages the global evaluation of the application performance. CPs are the modules which cooperate with it, by collecting events and preprocessing the information as possible. Thus, before evaluating the performance model, the Global Analyzer receives the relevant condensed information from the

CPs, in place of receiving every event from the application, as in the first implementation of MATE.

MATE : Monitoring, Analysis and Tuning Environment. It provides dynamic and automatic tuning of parallel applications. The performance knowledge used to tune the applications is based on performance models.

Optimization : this term is used in the broadest sense of the word to mean "*improvement*", i.e. we are not strictelly considering mathematical optimization.

Process : each program in execution. Parallel applications consist in a set of processed executing and cooperating in parallel. Sometimes, *process* is used as a synonym of *task* (except in Chapter 1).

Task : this word has three different meanings, which can be deduced from the context. In general terms (such as in Chapter 1), *task* is used to refer to some job or application to be executed. Another meaning of this word is related to a specific process in the application, i.e. some kind of the cooperating processes which executes a determined code; this is why sometimes *task* is used as synomym of *process*. Finally, this word is used to represent a subset of the total data to be processed, which represent a piece of job to be processed by some process.

Tuning technique : it is constituted by all the required information related to one particular problem, i.e. the application knowledge that represents specific, determined information about performance problems that can occur during application execution and solutions to these problems. In MATE, each tuning technique is implemented as a tunlet.

Tunlet : is a piece of software used by MATE where the knowledge about a particular performance problem is encapsulated. The main elements in a tunlet are the *measure points*, the *performance model* and the *tuning points/actions*. Tunlets provide the knowledge used by the Analyzer

(or Global Analyzer) process of MATE to require the monitoring instrumentation, evaluate the behaviour of the application and require for tuning actions.

Work : generally, this word is used to refer to some job to be executed or some data to be processed. It is related to *task*, due to in general terms both are used in a similar manner. Therefore, work is used to mean *a portion of data to be processed*, or to mean *a specific functionality of the program, i.e. a subset of sentences to be executed.* The meaning depend on the decomposition strategy under consideration: data parallelism or tasks parallelism.

Bibliography

[1] Aho, A., Sethi, R., Ullman, J., *Compilers - Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 1986.

[2] Aho, A., Ullman, J., *The Theory of Parsing, Translation, and Compiling - Volume 1: Parsing*, Prentice Hall, 1972.

[3] Armstrong, C.W., Ford, R.W., Gurd, J.R. , Lujan, M., Mayes, K.R., Riley, G.D., *Performance Control of Scientific CNC. Coupled Models in Grid Environments*, Concurrency and Computation: Practice and Experience, Special Issue: Grid Computing. Eds. J. Gurd, T. Hey, J, Papay and G. Riley. Vol. 17, No. 2-4, pp. 259-295, February-April 2005.

[4] Bala, V., Duesterwald, E., Banerja, S., *Dynamo: A Transparent Dynamic Optimization System*, Hewlett-Packard Labs, PLDI. Vancouver, 2000.

[5] Benkner, S., Andel, S., Blasko, R., Brezany, P., Celic, A., Chapman, B., Egg, M., Fahringer, T., Hulman, J., Hou, Y., Kelc, E., Mehofer, E., Moritsch, H., Paul, M., Sanjari, K., Sipkova, V., Velkov, B., Wender, B., and Zima, H., *Vienna Fortran Compilation System - Version 2.0 - User's Guide*, October 1995.

[6] Berman, F., Wolski, R., *Scheduling From the Perspective of the Application*, High Performance Distributed Computing 1996. Syracuse, NY, USA, August 1996.

[7] Buck, B., Hollingsworth, J.K., *An API for Runtime Code Patching*, University of Maryland, Computer Science Department. Journal of High Performance Computing Applications. 2000.

[8] Buyya, R. et al, *High Performance Cluster Computing - Architectures and Systems (Volume 1)*, Prentice Hall, 1999.

[9] Carey, G. et al, *Parallel Supercomputing: Methods, Algorithms and Applications*, John Wiley & Sons, 1989.

[10] Caymes Scutari, P., *Entorno de Desarrollo y Sintonización de Aplicaciones Master/Worker*, Universitat Autònoma de Barcelona, Departament d'Arquitectura i Sistemes Operatius. Master. 2005.

[11] Caymes-Scutari, P., Morajko, A., César, E., Mesa, J., Costa, G., Margalef, T., Sorribes, J., Luque, E., *Entorno de Desarrollo y Sintonización de Aplicaciones Master/Worker*, IX Congreso Argentino de Ciencias de la Computación CACIC 2005. Electronic Proceedings of the conference. 17/10/05 - 21/10/05. Concordia, Argentina. 2005.

[12] Caymes-Scutari, P., Morajko, A., Margalef, T., Luque, E., *Automatic generation of dynamic tuning techniques*, Euro-Par 2007. 28-31 August 2007. Laboratory IRISA, Rennes, France. *Accepted.*

[13] Caymes-Scutari, P., Morajko, A., Margalef, T., Sorribes, J., Luque, E., *Generación Automática de Técnicas de Sintonización Dinámica*, XVII Jornadas de Paralelismo-Albacete 2006. Proceedings of the XVII Jornadas de Paralelismo, p. 383-388. Universidad de Castilla-La Mancha, 18/09/06 al 20/09/06. Albacete, Spain.

[14] César, E., Mesa, J.G., Sorribes, J., Luque, E., *Modeling Master-Worker Applications in POETRIES*, IEEE 9th International Workshop HIPS 2004, IPDPS, pp. 22-30. April, 2004.

[15] Cesar, E., Moreno, A., Sorribes, J., Luque, E., *Modeling Master/Worker applications for automatic performance tuning*, Parallel Computing, Volume: 32, p. 568-589, September 2006.

[16] Chen, W., Lerner, S., Chaiken, R., Gillies, D.M., *Mojo: A Dynamic Optimization System*, Microsoft Research. 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO). Monterrey, California, December 2000.

[17] Crovella, M.E., LeBlanc, T.J., *The Search for Lost Cycles: A New Approach to Parallel Program Performance Evaluation*, Tech. Rep. 479, University of Rochester. 1994.

[18] Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., White, A., *Sourcebook of parallel computing*, Morgan Kaufmann Publishers. 2003.

[19] Espinosa, A., Margalef, T., Luque, E., *Automatic Detection of Parallel Program Performance Problems*, Lecture Notes in Computer Science, vol. 1573, pp. 365-377, Springer-Verlag. June 1998

[20] Fahringer, T., Zima, H.P., *A Static Parameter based Performance Prediction Tool for Parallel Programs*, 7th ACM International Conference on Supercomputing. Japan, July 1993.

[21] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V., *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Network Parallel Computing*, MIT Press. Cambridge, MA, 1994.

[22] Geist, A., Heath, T. M., Peyton, B. W., Worley, P. H., *A User's Guide to PICL: A Portable Instrumentation Communication Library*, TR TM-11616, Oak Ridge National Lab. 1990.

[23] Gerndt, M. et al, *Performance Tools for the Grid: State of the Art and Future. APART white paper*, Shaker Verlag, Research Report Series Vol.30. Aachen, 2004.

[24] Grama, A., Gupta, A., Karypis, G., Kumar, V., *Introduction to Parallel Computing*, Pearson Addison Wesley. Second Edition. 2003.

[25] Groop, W., Lusk, E., *User's Guide for mpich, a Portable Implementation of MPI*, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[26] Groop, W., Lusk, E., Doss, N., Skjellum, A., *A high-performance, portable implementation of the MPI message passing interface standard*, Parallel Computing, volume 22-6, pp.789-828, September 1996.

[27] Heath, M., Etheridge, J., *Visualizing the Performance of Parallel Programs*, IEEEComputer, vol. 28, pp. 21-28. November 1995.

[28] Jain, R., *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation and modeling*, John Wiley & Sons, 1991.

[29] Jordan, H.F., Alaghband, G., *Fundamentals of Parallel Processing*, Pearson Prentice Hall. 2003.

[30] Kale, L.V. and Krishnan, S., *Charm++ : A portable concurrent object oriented system based on C++*, Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications. September 1993.

[31] Krishnan, S., Kale, L. V., *Automating Parallel Runtime Optimizations Using Post-Mortem Analysis*, International Conference on Supercomputing, pp. 221-228. 1996.

[32] Maillet, E., *TAPE/PVM an Efficient Performance Monitor for PVM Applications – User Guide*, LMC-IMAG, Grenoble, France, June 1995.

[33] Mattson, T., Sanders, B., Massingill, B., *Patterns for Parallel Programming*, Addison-Wesley, 2004.

[34] Mayes, K.R., Elliot, M.J., Manning, A.M., Haglin, D.J. and Gurd, J.R., *A distributed search infrastructure for Statistical Disclosure Control on a Grid*, Proceedings of the Second International Conference on e-Social Science, Manchester, June, 2006.

[35] Mayes, K.R., Lujan, M., Riley, G.D., Chin, J., Coveney, P.V. and Gurd, J.R., *Towards Performance Control on the Grid*, Philosophical Transactions of the Royal Society of London Series A, Vol. 363, No. 1833, pp. 1793-1806, August 2005.

[36] Mesa, J.G., *Framework Master/Worker*, Universitat Autònoma de Barcelona, Departament d'Informàtica. Master. 2004.

[37] Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T., *The Paradyn Parallel Performance Measurement Tool*, IEEE Computer vol. 28. pp. 37-46. November 1995.

[38] Morajko, A., *Dynamic Tuning of Parallel/Distributed Applications*, Phd. Thesis. Universitat Autònoma de Barcelona. 2004.

[39] Morajko, A., Morajko, O., Jorba, J., Margalef, T., Luque, E., *Dynamic Performance Tuning of Distributed Programming Libraries*, LNCS, 2660, pp. 191-200. 2003.

[40] Morajko, A., Morajko, O., Margalef, T., Luque, E., *MATE: Dynamic Performance Tuning Environment*, LNCS, 3149, pp. 98-107. 2004.

[41] Morajko, A., Caymes-Scutari, P., Margalef, T., Luque, E., *MATE: Monitoring, Analysis and Tuning Environment for Parallel/Distributed Applications*, Concurrency and Computation: Practice and Experience. 2005. *Accepted.*

[42] Morajko, A., Caymes, P., Margalef, T., Luque, E., *Automatic Tuning of Data Distribution Using Factoring in Master/Worker Applications*, 5th Internacional Conference in Computational Science (ICCS2005). Part III (LNCS 3515), p. 132-139. 22-25. Atlanta, GA, United States. May 2005.

[43] Morajko, A., César, E., Caymes-Scutari, P., Mesa, J., Costa, G., Margalef, T., Sorribes, J., Luque, E., *Development and Tuning Framework of Master/Worker Applications*, Journal of Computer Science & Technology (JCS&T) - October 2005. Invited Paper. Vol. 5 No. 3. Págs. 115-112. Argentina. October 2005.

[44] Morajko, A., César, E., Caymes-Scutari, P., Margalef, T., Sorribes, J., Luque, E., *Automatic Tuning of Master/Worker Applications*, Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference. LNCS 3648, p. 95-103. 30/08/05 - 02/09/05. Lisboa, Portugal.

[45] Nagel, W., Arnold, A., Weber, M., Hoppe, H., *VAMPIR: Visualization and Analysis of MPI Resources*, Supercomputer 1 pp. 69-80. 1996.

[46] *Paradyn Project: Paradyn Parallel Performance Tools, User's Guide, Release 4.0*, University of Wisconsin, Computer Science Department. May 2003.

[47] Parker, S.G., Johnson C.R., *SCIRun: A Scientific Programming Environment for Computational Steering*, SC'95. San Diego, USA, December 1995.

[48] Reed, D.A., Roth, P.C., Aydt, R.A., Shields, K.A., Tavera, L.F., Noe, R.J., Schwartz, B.W., *Scalable Performance Analysis: The Pablo Performance Analysis Environment.*, Proceedings of Scalable Parallel Libraries Conference, pp. 104-113, IEEE Computer Society. 1993.

[49] Ribrel, R.L., Vetter, J.S., Simitci, H., Reed, D.A., *Autopilot: Adaptive Control of Distributed Applications*, High Performance Distributed Computing 1998, pp. 172-179. Chicago, August 1998.

[50] Stroustrup, B., *El lenguaje de programación C++*, Addison Wesley, 2001.

[51] Tanenbaum, A., Van Steen, M., *Distributed Systems - Principles and Paradigms*, Prentice Hall, 2002.

[52] Tapus, C., Chung, I-H., Hollingsworth, J.K., *Active Harmony: Towards Automated Performance Tuning*, SC'02. November 2002.

[53] Wilkinson, B., Allen, M., *Parallel Programming - Techniques and Applications Using Networked Workstations and Parallel Computers*, Pearson Prentice Hall. Second Edition. 2005.

[54] Yan, J., Sarukhai, S., *Analyzing Parallel Program Performance Using Normalized Performance Indices and Trace Transformation Techniques*, Parallel Computing, vol. 22, pp. 1215-1237. 1996.

Web Links

[55] *Active Harmony*,
http://www.dyninst.org/harmony/
Accessed on November 2006.

[56] *Automatic Instrumentation and Monitoring System (AIMS)*,
http://www.nas.nasa.gov/Software/AIMS/
Accessed on March 2007.

[57] *Distributed Applications and Middleware for Industrial Use of European Networks (DAMIEN)*,
http://www.hlrs.de/organization/pds/projects/damien/
Accessed on November 2006.

[58] *Document Object Model (DOM)*,
http://www.w3.org/DOM/
Accessed on September 2005.

[59] *European Center for Parallelism of Barcelona (CEPBA)*,
http://www.cepba.upc.es/
Accessed on November 2006.

[60] *European Center for Parallelism of Barcelona - Dimemas*,
http://www.cepba.upc.es/dimemas/
Accessed on November 2006.

[61] *Extensible Markup Language (XML)*,
http://http://www.w3.org/XML/
Accessed on September 2005.

[62] *Flex, a fast scanner generator*, Paxon, V.,
http://www.gnu.org/software/flex/manual/
Accessed on September 2005.

[63] *Java SE HotSpot at a Glance*,
http://java.sun.com/javase/technologies/hotspot/
Accessed on November 2006.

[64] *MIPCL Portable Instrumentation Library*, Worley, P. H.,
http://www.csm.ornl.gov/picl/mpicl.html
Accessed on November 2006.

[65] *ParaDyn - Parallel Performance Tools*,
http://www.paradyn.org
Accessed on November 2006.

[66] *ParaGraph: a performance visualization tool for MPI*,
http://www.csar.uiuc.edu/software/paragraph/
Accessed on November 2006.

[67] *Portable Instrumented Communication Library - PICL*,
http://www.epm.ornl.gov/picl/picl2.html
Accessed on November 2006.

[68] *Scientific Computing and Imaging Institute - SCIRun*,
http://www.sci.utah.edu/
Accessed on November 2006.

[69] *Vampir: Visualization and Analysis of MPI Resources*,
http://www.fz-juelich.de/zam/docs/autoren95/nagel2
Accessed on November 2006.

[70] *XPVM: A Graphical Console and Monitor for PVM*,
http://www.netlib.org/utk/icl/xpvm/xpvm.html
Accessed on November 2006.

[71] *XQuery 1.0, XPath 2.0, and XSLT 2.0 Functions and Operators*,
http://www.w3.org/2005/04/xpath-functions
Accessed on October 2005.

[72] *XSL Transformations (XSLT) - Version 1.0*,
http://www.w3.org/1999/XSLT/Transform,
http://www.w3.org/TR/xslt
Accessed on October 2005 and March of 2007, respectively.