SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2012; **42**:281–302
Published online 9 March 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1061

A methodology for transparent knowledge specification in a dynamic tuning environment

P. Caymes-Scutari[1,*,†], A. Morajko[2], T. Margalef[2] and E. Luque[2]

[1]*Laboratorio de Investigación en Cómputo Paralelo/Distribuido, Departamento de Ingeniería en Sistemas de Información, Universidad Tecnológica Nacional—Facultad Regional Mendoza, Rodríguez 273, (M5502AJE) Mendoza, Argentina—CONICET (Argentina)*
[2]*Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona, Edifici Q—Campus, Bellaterra, 08193 Barcelona, Spain*

SUMMARY

The increasing use of parallel/distributed applications demands a continuous support to take significant advantages from parallel power. This includes the evolution of performance analysis and tuning tools which automatically allows for obtaining a better behavior of the applications. Different approaches and tools have been proposed and they are continuously evolving to cover the requirements and expectations of users. One such tool is MATE (Monitoring Analysis and Tuning Environment), which provides automatic and dynamic tuning for parallel/distributed applications. The knowledge used by MATE to analyze and take decisions is based on performance models which include a set of performance parameters and a set of mathematical expressions modeling the solution of the performance problem. These elements are used by the tuning environment to conduct the monitoring and analysis steps, respectively. The tuning phase depends on the results of the performance analysis. This paper presents a methodology to specify performance models. Each performance model specification can be automatically and transparently translated into a piece of software code encapsulating the knowledge to be straightforwardly included in MATE. Applying this methodology, the user does not have to be involved in the implementation details of MATE, which makes the usage of the tool more transparent. Copyright © 2011 John Wiley & Sons, Ltd.

Received 15 October 2010; Revised 30 December 2010; Accepted 10 January 2011

KEY WORDS: specification; automatic development; performance model; parallel/distributed computing; automatic performance analysis; dynamic tuning

1. INTRODUCTION

In the recent years, the demand for high-performance computing has increased significantly. Scientists have to solve complex problems involving large volumes of data with complex operations over them. In this context, parallel/distributed computing has emerged to provide the computational power required to overcome these problems. However, developing parallel/distributed applications is a difficult task since it requires the usage of parallel programming models and communication libraries. Moreover, it involves some additional aspects, such as synchronization, concurrency, scalability, and decomposition, which determine the correct behavior of the application [1]. In addition, the performance of applications is a crucial issue, and therefore a

*Correspondence to: P. Caymes-Scutari, Laboratorio de Investigación en Cómputo Paralelo/Distribuido, Departamento de Ingeniería en Sistemas de Información, Universidad Tecnológica Nacional—Facultad Regional Mendoza, Rodríguez 273, (M5502AJE) Mendoza, Argentina.
†E-mail: pcaymesscutari@frm.utn.edu.ar

Copyright © 2011 John Wiley & Sons, Ltd.

high degree of expertise is required to program an efficient application that takes full advantage of parallelism. Furthermore, given the frequent heterogeneity and/or the dynamic characteristics of the systems, even experts in the field have to face the optimization or tuning process with the aim of improving the application and its behavior. The above mentioned determines different kinds of parallelism users, depending on their knowledge and experience in the field. This determines the necessity for counting with tools covering the different abstraction levels. Fortunately, along the years different approaches have been proposed and many tools have appeared to assist users in some phase of the optimization process: monitoring, performance analysis, or tuning. Here we summarize some of these tools showing the experience that is required from a user.

First, a variety of monitoring and visualization tools appeared [2–5]. However, although they facilitated the task of the user in collecting information, the user had to be responsible for the analysis of such data to make decisions in how to improve the application. Then automatic analysis tools appeared in order to exempt the user from being an expert in performance analysis [6, 7]. Finally, the tuning tools tended (and they still do) toward the automatic and dynamic introduction of changes in the application without the users' intervention [8]. Although the tools intend to make the users' task easier, the analysis is based on a single execution of the application and the tuning changes are post mortem, preparing it to improve future executions. Thus, decisions made as a consequence of performance analysis are only useful in case the application is always executed under the same conditions: homogeneous system, data (application input) and/or exclusive system resources.

The dynamic tuning approach is especially suitable for heterogeneous or time-sharing execution environments, due to monitoring, analysis, and tuning are executed on the fly with the aim of adapting the behavior of the application to the current conditions of the system. There exist different tools implementing this approach [9–11]. The main differences among them are related to the monitoring and tuning technology, and the knowledge representation used to analyze the performance of the application: fuzzy logic, heuristics, history, or performance models. In this paper we focus on MATE (Monitoring, Analysis and Tuning Environment) [12–14], a dynamic and automatic monitoring, analysis and tuning environment based on performance models. MATE provides certain advantages when tuning an application: the application does not have to be reimplemented with extra source code for the monitoring purposes since MATE automatically instruments the application on the fly. Moreover, the performance analysis is relatively concise because it is based on the evaluation of a series of mathematical expressions rather than on a complex kind of search.

The knowledge represented by each performance model is encapsulated in MATE as a '*tunlet*', i.e. a piece of software which condenses the logic to tune a particular performance problem, indicating what to monitor in the application, how to evaluate the collected information, and what to change, when, and where. Although MATE and other dynamic tuning tools are very useful, the lack of transparency is common to all of them. In general, the user has to get involved with the techniques and technologies used by the tool, whether for directly instrumenting the application (Active Harmony [10], Autopilot [9]) or for preparing the instrumentation process (PerCo [11] and MATE). In the particular case of MATE, until now, the inclusion of knowledge has been manual and dependent on the implementation of the tuning environment. This restricted the usage of MATE since users had to know its implementation details. In this paper we concentrate on the transparency of the performance model specification in MATE. We present a methodology proposed to make easier the task of including the performance models into MATE. We defined a set of abstractions to represent the elements of the tunlet and a series of steps to conform the corresponding specification. We defined a tunlet specification language and developed a translator to automatically create a tunlet from a specification.

In the following section we document MATE in more detail. Section 3 presents the proposed methodology to transparently include knowledge in MATE. Section 4 shows a complete simple example of the usage of this methodology. Section 5 presents results related to the effectiveness

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

of MATE and the usefulness of the proposed methodology to automatically create tunlets. Finally, Section 6 draws the conclusions.

2. MATE

MATE is an environment which provides dynamic and automatic tuning for iterative parallel/ distributed applications. The steering of the application comprises three different phases: monitoring the behavior of the application, performance analysis using the collected information, and tuning of the application. 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 changing conditions of the system. In this way, the user is exempted from manual application tuning. MATE includes several components which cooperate among them to control and improve the execution of the application. The main components are the following:

1. *Application Controller* (*AC*) is a daemon-like process which controls the execution of individual application tasks. It is responsible for monitoring (dynamic instrumentation and creation of events) and tuning. These functionalities are implemented by the *Monitor* and the *Tuner* modules, respectively.
2. *Dynamic Monitoring Library* (*DMLib*) is a shared library, which is dynamically loaded in the application tasks. It is used to perform data monitoring and collection.
3. *Analyzer* carries out performance analysis of the application. In addition, it decides what have to be monitored (to obtain the necessary metrics to evaluate the performance) and tuned (to improve the behavior of the application according to the conclusions obtained after the performance analysis). From a functional point of view, the Analyzer is divided into two main parts:
   (a) *Dynamic Tuning API* (*DTAPI*) which constitutes the interface of the Analyzer module to communicate with the Monitor and Tuner modules. DTAPI provides the Analyzer with a global view of the application, the tasks and the events, and encapsulates all the low-level issues related to controlling the execution of the parallel application.
   (b) *Tunlets*, where each 'tunlet' may be defined as a software component which describes a particular performance problem of a running application. It provides the logic that indicates how to evaluate the behavior and modify the execution to improve the application performance. Each tunlet should use the DTAPI to allow the Analyzer module to communicate with AC and DMLib. The Analyzer is responsible for managing the application by invoking monitoring and tuning actions that are provided by a tunlet. Tunlets constitute the core of the tuning approach of MATE in terms of representation of knowledge.

The collection and processing of monitoring data is carried out in a distributed-hierarchical manner to avoid the analysis process from being a performance bottleneck. In this way we also go toward making MATE scalable. This approach is explained in detail in [14].

In addition to its tuning properties, MATE is provided with a framework for the development of Master/Worker applications [15, 16]. MATE provides a set of ready-to-use tunlets defined in terms of framework-related performance problems (the measure points, performance functions, and tuning points depend on the framework classes). Therefore, every application developed using this framework can be automatically tuned by those tunlets. This represents a great benefit especially for non-expert users: they can develop their Master/Worker applications considering high-level abstractions, without entering into communications and synchronization details. Moreover, they can also automatically tune these applications according to the problems the tunlets can overcome. At the moment, MATE offers only the Master/Worker framework [17], but in the future, we expect to incorporate new programming models, skeletons and tunlets to tune a wider range of parallel applications.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

When MATE is used, the run-time changes of the application, for both the monitoring and tuning processes, are implemented via the dynamic instrumentation library DynInst [18]. MATE and its approach have been previously presented in detail in [12, 14].

2.1. Tunlets

Tunlets are the core of the dynamic and automatic tuning approach implemented by MATE as they provide the knowledge required to improve the performance of parallel applications. Each tunlet defines and implements a particular tuning technique, i.e. the logic to overcome a particular performance problem encapsulating the knowledge about this problem. The general structure of a tunlet includes

1. *Measure points* which indicate *what* must be measured in the application to be able to evaluate its behavior. This definition includes values of variables, parameters, function returns, timestamps, etc.
2. *Performance functions* are mathematical expressions which determine *how* to evaluate the collected information in order to detect bottlenecks.
3. *Tuning points/Actions/Synchronization* indicate *what*, *where*, and *when* to change in the application execution with the aim of adapting its behavior.

At this point, an obvious question is: how to determine a performance model? In general, parallel applications follow a parallel paradigm, such as Master/Worker or Pipeline [19]. This fact provides us with an advantageous situation as for each parallel paradigm we can indicate a set of well-known performance problems. In the case of the Master/Worker paradigm, it presents two serious problems: *load balance* which depends on the characteristics of the environment where the application is executed and on the applied assignment strategy (complete or on demand) and the *number of workers*, which depends on the volume of data to be processed, the computational power and the communication cost. In the case of the Pipeline paradigm, it presents problems related to load balancing. Every stage in the pipeline may have a different volume of computation, and in consequence the throughput is determined by the slowest stage of the pipeline.

Such kinds of performance problems are already modeled, for example in [20–22]. Therefore, a person who wants to analyze the performance of the parallel application can use these already available mathematical models or he/she can develop a new particular performance model, if necessary. In both cases, the considered model may constitute a piece of knowledge that can be included as a tunlet in MATE.

As an example of a performance model, we consider the one presented in [20] that defines a tunlet for a Master/Worker framework associated with MATE. This performance model offers the means to calculate the optimal number of workers for a Master/Worker application:

$$Nopt = \sqrt{\frac{\lambda V + Tc}{tl}} \tag{1}$$

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. The expression is defined as a function of computing time (Tc), total data volume (V), latency (tl), and bandwidth (λ). For this example, Tc, V, tl, and λ constitute the performance parameters of the model which can be calculated using the measurements provided by the inserted instrumentation. The application is instrumented in different points that represent the variables and values which allow the collection of the required information. Given that MATE is based on event tracing, the measure points are captured and associated to events along the execution time. For example, in the case of V, it is calculated as $\sum(v_i + v_m)$, where v_i represents the size of the tasks sent to each worker$_i$, in bytes, and v_m represents the size of the answer sent back to the master for each worker, in bytes. For this model, if the application is not developed with the framework provided by MATE, the measure points depend on the implementation and the names of the variables in the application. Considering the names of the variables, functions, parameters, etc. of the framework source code, the value of v_i can be obtained from the variable

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

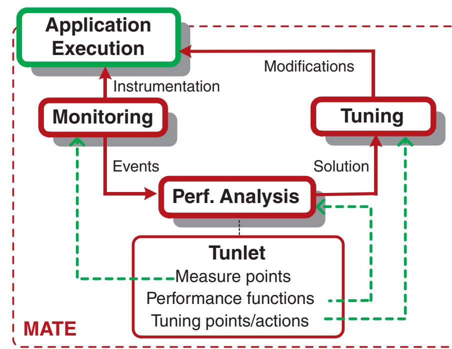

Figure 1. Basic operation of MATE based on the tunlet knowledge.

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*. In this case, v_i must be captured when the master sends tasks to the workers. 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.

Once the application is developed, the performance model determined, and the tunlet prepared, the application may pass to the tuning process. Figure 1 illustrates the execution of the application under MATE. When the execution of the application under MATE starts, the tunlet indicates to the Analyzer the set of measure points that are required by the model. The Analyzer forwards the requirement to every AC (distributed over processors where the application is being executed). Consequently, the corresponding instrumentation is automatically inserted online in the code to capture events as the entry of send and receive functions. Such events will also contain the values of *TheWorkUnitBytes*, *NumTuples*, and *NBytes* to calculate V. Then, during the execution of iteration i the Analyzer receives event records related to such iteration generated by different processes and the tunlet is notified. The tunlet classifies and examines the set of received event records, with the aim of extracting the necessary measurements. When all the information about iteration i has been received and processed, the tunlet evaluates the performance functions according to the behavior of the application along the iteration i. In the example, the tunlet evaluates the Expression (1) to determine the optimal number of workers according to the current conditions. If the tunlet detects a performance bottleneck, it decides if the performance can be improved considering the existing conditions. If so, the tunlet informs the Analyzer about the possible improvement. Consequently, the Analyzer requests 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, for being managed by the Tuner. In the example, the tuning action will indicate that the value of the variable controlling the number of workers needs to be changed. The tuning actions required by the tunlet will improve the execution of the following iterations. In general, the execution of the application and the tuning actions of MATE are not explicitly synchronized, meaning that the application does not stop its execution (although there are some cases in which the user can determine a synchronized tuning at the expense of the idle time waiting for the insertion of the modifications), but MATE is changing the application on the fly. Consequently, when the tunlet makes a decision regarding iteration i, the application is executing the iteration $i+1$, and the improvement of the tuning action will be effective at iteration $i+2$. The dashed arrows in the rounded dashed square (tagged 'MATE') of Figure 1, conceptually show the elements required directly by the monitoring, analysis, and tuning phases provided by the tunlet.

2.2. Inclusion of the knowledge in MATE

Given that the purpose of the paper is to present a methodology for transparent knowledge specification, it is important to indicate how the performance knowledge inclusion was previously

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

managed. MATE may tune performance problems at different layers and consider diverse tuning approaches. The provided possibilities of the usage of MATE determine a trade-off between level of abstraction, benefits, ease of use, and required user knowledge. With respect to the tuning layers, we have distinguished different layers of the application that can be tuned: application-specific code, standard and custom libraries (API+code), operating system libraries (API+code), and use of hardware resources. For some layers we have many common information and hence we can extract well-defined bottleneck representatives for many applications and define their solutions. In other cases, it is required to provide the knowledge about the specific application problems and solutions since there is no information about the potential application bottlenecks. Note that due to incomplete application information, dynamic tuning of unknown applications is difficult, or sometimes even impossible. Generally, the performance analysis might not be performed effectively without knowledge about what the application does. Dynamic modifications of unknown application structures are complex, may appear as dangerous and hence, must be done very carefully. In other words, tuning of the application code is the most complex task, due to the lack of problem-specific knowledge. Each application-specific implementation may be totally different and there might be no parts common for many applications although they provide the same functionality. The upper the layer, the more specific the information about the application required, and conversely, the lower the layer, the more generic the information available. Summarizing, this constitutes the main motivation of this work: the necessity of simplifying and making transparent the use of MATE when a specific problem has to be tuned.

With respect to the tuning approaches, MATE offers mainly two of them:

- *Automatic approach*: When there is no available information about a particular application, just generic performance problems may be treated. In this case, the tunlets are predefined and included in MATE, particularly considering information non-dependent on any specific application. In this approach the user is not involved in the tunlets preparation/programming but he/she only uses them. Thus, this approach is very suitable for non-expert users in performance modeling and tuning, at the expense of being constrained to a determined set of predefined tuning techniques. The generic performance problems considered may rise at different library levels. In this case, the often tackled problems are related to the operating system or library levels. From a more specific point of view, when the applications are developed using the framework included in MATE for Master/Worker applications, the typical problems of Master/Worker parallel programming model—such as problems with communications, synchronization, and decomposition—can also be tackled. Figure 2(a) illustrates this approach representing the usage of MATE as a black box. The user is only responsible for the application development, but he/she does not have to worry about the performance model determination nor the tunlet creation.
- *Cooperative approach*: More specific problems can be tuned when the user provides information about the application. In this case, the user must prepare the application for the tuning process (e.g. re-declare certain variables as global or write some part of the code as a function) and define an application-specific knowledge to model what to measure, how to evaluate it, and what to change to improve the execution time. Therefore, the user has to develop the corresponding tunlet, i.e. to implement the performance model that tunes the problems of the application. This approach is much more flexible than the previous one, given that any problem can be tackled. However, this flexibility requires a high degree of expertise in parallel programming and developing performance models. Moreover, the user has to provide the performance knowledge as a tunlet, which implies being familiar with the implementation of MATE, in particular with the DTAPI that a tunlet must use to be incorporated in MATE. In Figure 2(b) we illustrate this approach representing the involvement of the user in providing MATE with new knowledge.

Each approach has advantages and specific constraints that determine the degree of flexibility in the usage of MATE. The ideal solution would be to combine direct and transparent usage of MATE with the flexibility to incorporate the knowledge for the treatment of any performance

Copyright © 2011 John Wiley & Sons, Ltd.

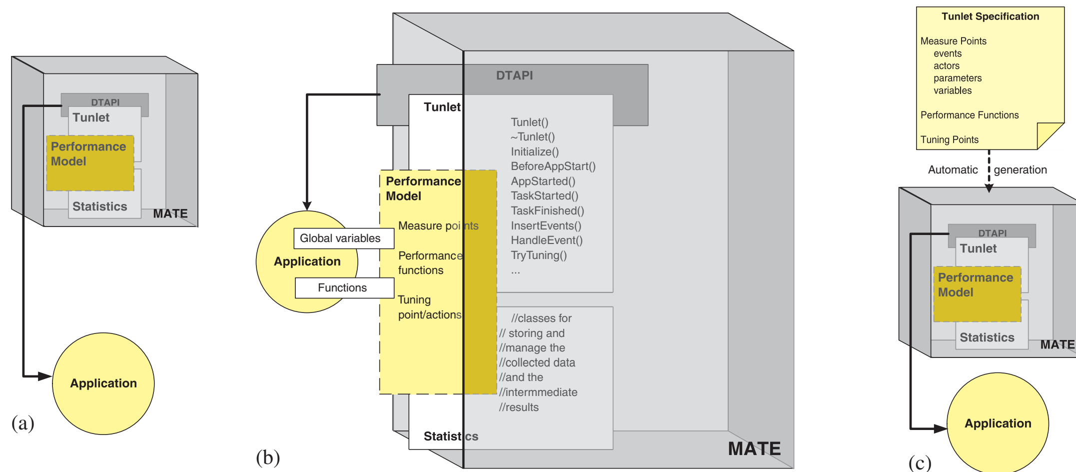

Figure 2. Different approaches to use MATE: (a) automatic approach; (b) cooperative approach; and (c) cooperate to automate approach.

problem. To achieve this goal, it is necessary to have an abstraction mechanism that works as the interface between the information possessed by the user and the MATE environment. The implementation details of MATE should be hidden and therefore made transparent for the users, which would relieve them from the programming responsibility. In Section 3, we present a methodology proposed to provide such an approach, making easier and transparent the cooperation between the users and MATE. The complete methodology provides the easier knowledge representation, automatic creation of tunlet code, incorporation into MATE, and reusability for other applications.

3. TRANSPARENT INCLUSION OF KNOWLEDGE IN MATE

In this section we present a new approach to include performance knowledge in MATE: the '*cooperate to automate approach*'. It includes a methodology to automatically develop tunlets from a given specification and constitutes an interface between the user and MATE in which the implementation details are transparent. In Figure 2(c) we illustrate this new approach. As in the automatic approach, MATE is used as a black box, the user does not directly use DTAPI to create a tunlet. However, the user is responsible for providing the information related to the application and to the performance model necessary to automatically create the tunlet. Clearly, it demands a cooperation of the user—similar to the cooperative approach. However, the advantage is that the user works at an abstract and descriptive level rather than at a programming and implementation level. Figure 3 extends the previously explained approach, indicating the user's point of view. Let us suppose a particular parallel application with some specific performance problem. In many cases it is possible to overcome the problem or reduce its negative effects in order to improve the application performance. As mentioned in Section 2, MATE uses an effective and fast approach based on performance models, i.e. a set of expressions describing the performance problem. As the figure shows, the user only needs to determine the performance model to be specified and he/she has to provide the necessary information about the parallel application in order to define the elements required by the specification of the corresponding tunlet. Once the tunlet specification is complete, the creation of the tunlet is performed automatically. Therefore, the performance problem can be tuned by using the tunlet incorporated in MATE. In summary, the user has to *cooperate* to *automate* the creation and use of the tunlet.

In the subsequent subsections, we present the abstractions utilized to specify a tunlet, the methodology to provide a tunlet based on a performance model, the defined tunlet specification language, and the tunlet generator. For simplicity reasons, in Section 4 we provide only a simple

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

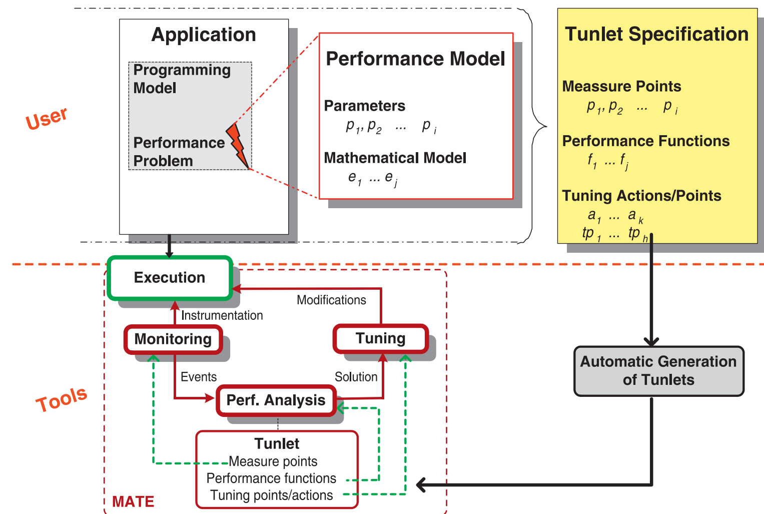

Figure 3. Development of a tunlet basing on its specification.

example of the usage of this methodology. Results related to the example presented in Section 2.1 are described in Section 5. More details of the experimental results of the methodology can be obtained from [13].

3.1. Abstractions and terminology

The basic terminology used here is related to the philosophy of MATE in terms of functioning and knowledge representation: MATE acts to overcome *performance problems*. Fortunately, there is a set of well-known performance problems which are mathematically modeled. The *performance models* define the optimal behavior of the application and are mainly defined by two elements. On the one hand, the *performance parameters* which are needed to evaluate the expressions that represent the behavior of the application; these are the 'mathematical' variables involved in the evaluation of the performance functions. On the other hand, the *performance functions*, which express how to evaluate the performance parameters. In general, the results of the performance functions are used to determine the solution to the existing problem, i.e. how to improve the application execution time.

The three elements that each tunlet must provide—measure points, performance functions, and tuning points/actions—are defined considering the performance model of the problem to solve and the corresponding information about the application to interpret the parameters and functions of the model. Therefore, to provide the required knowledge to MATE and to be coherent with the DTAPI we define the following abstractions (see Figure 4):

1. *Actor*: In general, each 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, constitutes a different actor. For instance, in the Master/Worker model, master and worker are two different actors.
2. *Event*: It is the mechanism used by MATE in order to collect information about the application behavior. Events are captured at the entry or exit points of functions and they provide all the required information, for instance, timestamp, event place, and data volume.
3. *Variable*: It is a global variable of the application. A variable may be needed to obtain its value or change it.
4. *Value*: It is a value assumed by a parameter or the result of a function. Similar to the variables, values are useful to obtain information or to change such values.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

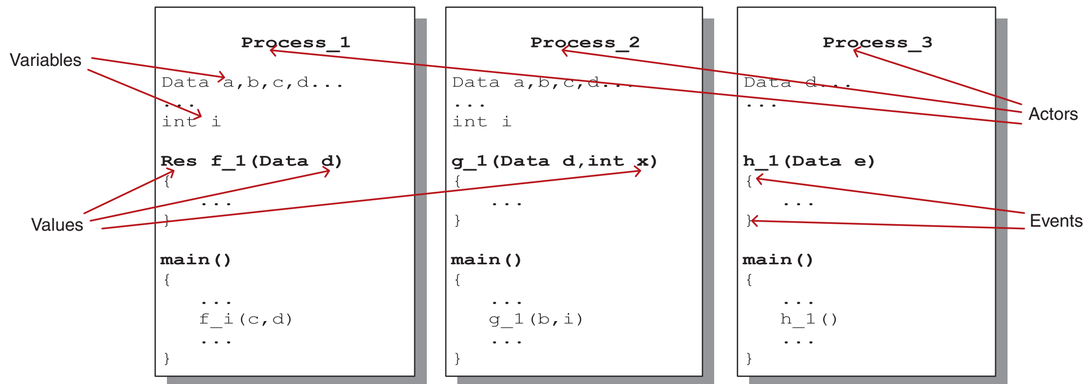

Figure 4. Abstractions in the application.

5. *Attribute*: It is a piece of information related to a particular entity. Actors and Events are the entities that may have a set of associated attributes. These attributes represent information related to the entity, which is used to determine the value of performance parameters. For instance, let suppose we want to know the computing time of each actor in a Master/Worker application. Then the master and each worker have to be provided with an attribute '*ct*' for storing the corresponding computing time. An example of an attribute associated to an event could be the timestamp when it occurred.

3.2. Methodology

Once the abstractions required for the knowledge provision have been defined, we briefly present the methodology a user should follow in order to define a tunlet. The methodology includes a series of steps to identify or/and interpret the previously defined abstractions in a parallel application and in a performance model utilized to improve the application behavior.

1. *Providing a performance model*. Given an application with some performance problem, the user has to determine how to model such a problem. The performance model may be a previously defined one or can be an *ad hoc* model defined by the user. In the first case, if the already existing model has been included in MATE, the user has to reinterpret such a tunlet according to the application, or he/she can directly use the tunlet in case the application was developed using the Master/Worker framework included in MATE. In the second case, i.e. 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, she/he has to be conscious of the 'collateral' problems the parallel application may present, and should be willing to dominate a certain set of concepts related to the parallel paradigm.
2. *Understanding the performance model*. The understanding of the model is a basic requirement as the model has to be interpreted according to the application. The user should concentrate especially in the study and in understanding the relevance and semantic of each performance parameter. As mentioned before, the performance functions are defined over the parameters; then if the user understands the semantic of each performance parameter, he/she will be able to interpret them in the application, and hence, the performance functions will be provided with all the required measurements to evaluate the application behavior.
3. *Interpreting the performance model*. This step—interpretation of the model—and the following one—identification of the actors—are closely related and interdependent, and hence, they could be done in parallel. The user has to determine the entities in the application that embody each performance parameter, i.e. how to provide each parameter with its semantics. In this step the user has to define the events to be captured and the information associated to them.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

(a) *Identifying the information/variables/values*. According to the semantic of each parameter, the user has to determine how to obtain its value, which events correspond to it and what kind of information they should contain. A special consideration is needed with variables: those variables that are read to obtain their value or to be changed, have to be global variables. Thus, this may require some adaptations in the implementation of the application, redefining variables as global, as well as using auxiliary global variables.
(b) *Identifying the events*. According to the semantic of each performance parameter (or some group of parameters), the user has to determine the entry and exit points of functions that have to be caught, i.e. the points in the application code that will provide the required information. Sometimes—especially when the required information is related to timestamps—the user will be forced to encapsulate some parts of the functionality of the application into functions in order to set the limits of the required points of execution.

4. *Identifying the actors in the application*. The user has to abstract the actors involved in the application, i.e. the different kinds of processes cooperating during the execution.

3.3. Tunlet specification language

In the previous sections we defined a set of abstractions and a methodology to specify a tunlet based on the performance model knowledge required by MATE. To help a user in a tunlet preparation process, we propose a mechanism to formalize such a conceptual definition of a logic about the possible solutions to a performance problem and to transform it into a tunlet. Therefore, we define a Tunlet Specification Language that helps users to develop a tunlet without entering into the implementation details of MATE. However, this language must consider the fact that MATE provides its DTAPI to incorporate the required knowledge. Summarizing Section 2, tunlets have to obey DTAPI, and hence the specification language must cover all the necessary elements to automatically generate a tunlet from a specification. In this paper we present the language in a general way, complementary details are documented in [13]. Our study on how to define the Tunlet Specification Language has been centered in the following aspects:

1. *How to capture the information*. We had to consider the methods provided by DTAPI to instrument the application, in particular, the properties which define an event.
2. *How to manage the collected information*. When an event is inserted in a process, a *handler* has to be determined to manage the event when received by the Analyzer.
3. *How and where to store the information*. Each tunlet manages a data structure for each iteration. In such a structure, the information collected is stored according to the nature of the information: information about actors or iteration.
4. *How to modify the application*. Similar to the insertion of monitoring instrumentation, we had to consider the methods provided by DTAPI to introduce modifications in the application, in case a tuning action is required.

The information encapsulated in a tunlet comes from two different and complementary sources: a *performance model* which provides the knowledge to tune a performance problem and the *information about the application* to interpret such a performance problem. Thus, the performance model determines three sections in the specification: *measure points*, *performance functions*, and *tuning points/actions*, while the application information provides the components to embody the performance model. From the point of view of the application we need to be aware of

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

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

In this way, we divided the specification into three different sections, which we describe in the following subsections.

3.3.1. Measure points. The *measure points* section embodies the largest part of the specification, as it condenses all the information about the application, the programming model and all the parameters of the performance model. The user must define:

1. The *actors* of the programming model. The tunlet needs the information about the actors of the application because each type of process may have different instrumentations inserted depending on the nature and role in the programming model. According to DTAPI, when a process is registered in the Analyzer, the tunlet must provide the location of the points in the code and discriminate what kind of instrumentation should be inserted. Therefore, it is necessary to declare the *name* of the actor, the *class* in which it is included or defined, and the name of the *executable file*. Moreover, some additional information is required from the tunlet point of view:
   (a) The *minimal* and *maximal quantity* of this type of actor that can be executed. It is needed to generate the structures that manage the behavioral information of each process during the successive iterations.
   (b) A *completion condition* to detect when the actor reached the end of its work during an iteration. This is necessary to check if every process finished before evaluating the performance functions.
   (c) The *actor's attributes*, i.e. the properties that 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.
2. The *variables* and *values* which can be instrumented or tuned in the application. For each of them a user must declare: the *name* and the *data type*, according to the declaration in the application; the type of the element: a *variable*, a *parameter*, or a *function output*; and the *actor* that has visibility of it. In general, these variables and values are used in defining the attributes of the specification elements as events and actors. The details (type, actor, etc.) are needed to locate them in the code and transmit them correctly.
3. The *events* to capture, such as entry or exit points of functions. Each event is defined by its *name*, the *actor* it is associated with, the *place* in the source code (*entry* or *exit* point of a function) and a code in case the event must be used to control the beginning or the end of an iteration. For each event it is also necessary to determine its *utility*, i.e. if the event must be always caught or if it is an addable or removable event. This is because some instrumentation could be added or removed from the application according to the current conditions of the system. Thus, utility may assume three different values: always, addable, removable. Certain *attributes*—*variables* or *values* measured when an event occurs—may be associated to a particular event. For example, the quantity of bytes sent can be a required metric generated when an event for the exit of a sending function occurs.
4. The *parameters of a model* that are treated as attributes of the performance model, whose values are generally calculated as a certain function of the attributes of actors or events. Although these parameters could be omitted in the tunlet, because the performance functions may obtain the values directly from the attributes of the entities, it is convenient to respect the parameters of the performance model to avoid mistakes in the interpretation.
5. As MATE was designed to tune iterative applications, it is necessary to indicate to which iteration the collected information is associated. Communications could cause a gap between the instant in which the information is sent and the moment in which it is received by the Analyzer. Therefore, to avoid inconsistencies, we require an additional section in the specification to collect information about each iteration. It includes an attribute to indicate the current iteration, and then all the additional information necessary to describe the behavior of each iteration, according to the performance model.

Copyright © 2011 John Wiley & Sons, Ltd.

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 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.

3.3.2. Performance functions. Considering the performance functions, they must be defined in C/C++ language. The user writes the functions' code using C/C++ and these functions will be recognized as the performance functions of the tunlet. Each function will be the value assigned to a tuning point or to an intermediate calculation. Any library needed to implement the functions should be declared in the *include* section.

3.3.3. Tuning actions/points. There could be different tuning actions to modify the behavior of the application: to modify the value of a certain variable in a determined process (`SetVariableValue`), to replace every call to a certain function by a call to another function (`ReplaceFunction`), to insert a new function call with its attributes (`InsertFunctionCall`), to eliminate every call to a certain function (`RemoveFunctionCall`), to call a certain function once during the execution (`OneTimeFuncCall`), and to change the value of a parameter in the entry of a function, before the body of the function is executed (`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 apply the tuning action.

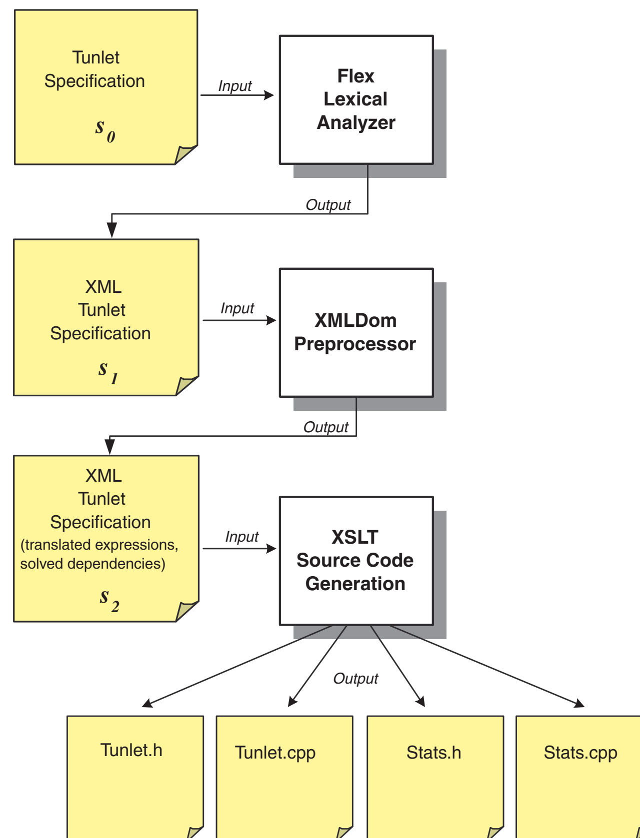

Figure 5. Phases in the generation of a tunlet from a specification.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

3.4. Tunlet generator

Once the tunlet has been specified, it is possible to translate such specification into source code. As part of this work, we developed a translator that supports the creation of a determined tunlet from a specification. The translation process to obtain the source code of the tunlet includes several steps, which are illustrated in Figure 5 and is briefly described here.

Let s_0 be the specification of a tunlet written by the user. In the first place, during the *lexical analysis*, s_0 is translated from its original text format into an equivalent XML [23] specification (s_1). Second, s_1 is *preprocessed* to solve the dependences among attributes and events through the specification with the aim of obtaining a new XML specification (s_2) without inconsistencies. In the last step, s_2 is used as the input of the *source code generation* process. The information to conform the source code of the tunlet is obtained from the different sections of s_2. We implemented this translator using several tools: Flex [24] for the lexical analyzer, an XMLDom program [25] for the preprocessor, and several XSLT stylesheets [26, 27] for the source code generator.

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

4. SIMPLE EXAMPLE

In order to clarify the functioning of the proposed methodology, we present a simple example of a tunlet specification. Let us suppose that we have the parallel application presented in Figure 6. The application consists of two kinds of processes: *Process_1* (P_1) and *Process_2* (P_2). Figure 7 shows a diagram of activities between both kinds of processes. Dashed lines represent the time line. Every iteration starts with a brief period of initialization. After that, P_1 divides the set of data to be processed into two parts. One portion is sent to P_2 and the other portion is processed by P_1. When P_2 finishes the reception of the data, the computing phase starts. When all the data have been processed, the results are sent back to P_1. After a brief phase of finalization of iteration, for example to gather all the results, it starts the next iteration.

Let us suppose now that the performance of this application is poor because the application execution lasts too long. The user may want to analyze how much time is wasted in communications.

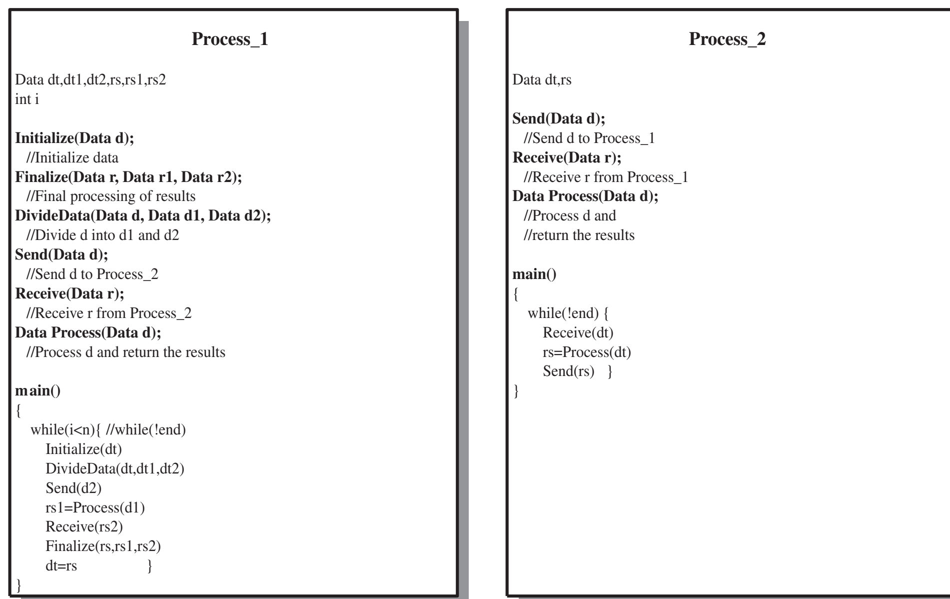

Figure 6. General view of a parallel application.

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

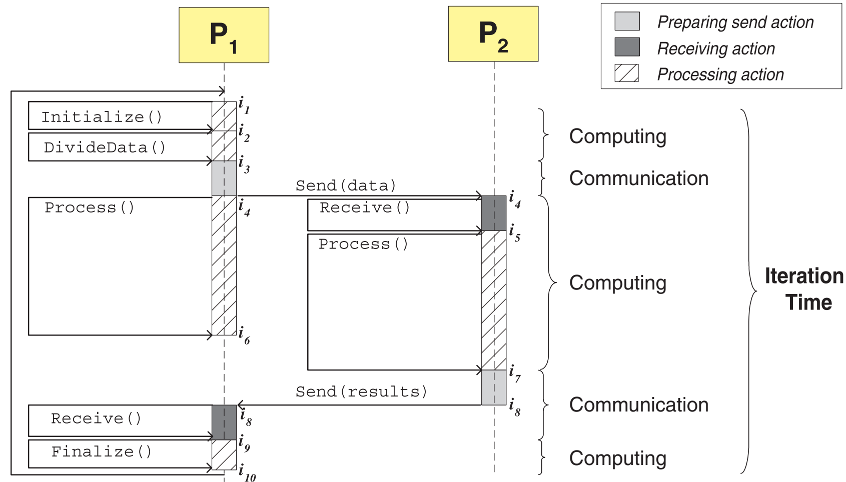

Figure 7. Diagram of activities and iteration time calculation.

Then, the user has to provide the mathematical model of the execution time of an iteration. Note that for this example we consider a slight difference between 'mathematical model' and 'performance model'. In Section 2, we introduced the three elements conforming a performance model: the measure points, the performance functions, and the tuning points and/or actions. The evaluation of expressions of performance in terms of measuring points can decide the appropriate tuning action to solve the problem. To simplify this example, we will not consider the various possibilities to overcome the bottlenecks nor improve the execution time. In other words, we are not considering the tuning points and/or actions, but we only analyze the time spent in computing and communications. It is clear that the analysis is not enough to improve the application performance if the program is not modified accordingly. This is the simple reason why we talk about mathematical model instead of model performance. In the following subsections we analyze this example.

4.1. First and second steps: Providing and understanding the model

In a scheme like the one presented in Figure 7, it is possible to calculate the total time of an iteration by adding the computing time and the communication time of such iteration. Let us suppose that $T_{foo}(x)$ represents the time spent during the iteration x to execute the foo function.

The execution time of the iteration it can be determined as follows: $T_{Ex}(it) = T_{Comp}(it) + T_{Comm}(it)$, where $T_{Comp}(it) = T_{Ini}(it) + T_{Proc}(it) + T_{Fin}(it)$ and $T_{Comm}(it) = T_{Send}(it) + T_{Recv}(it)$ where

- T_{Ini} comprises the initial treatment and configuration, previous to the parallel processing. Similarly, for T_{Fin} we consider the final treatment made over the processed data.
- T_{Proc} comprises the time in which the processes are processing the data.
- T_{Send} comprises the time spent on sending messages, whenever this time is not overlapped with computing time.
- T_{Recv} is the time spent on receiving messages.

In order to effectively 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. Let $Entry(foo)$ and $Exit(foo)$ be the functions to obtain the initial and final instants of a function foo. Considering this particular example, the mathematical model may be interpreted and expressed as shown in Figure 8. Although in this complete example we provide the model, some preexisting model could be used. In general, the use of a preexisting model provides more sense to the second step of the methodology. Clearly, in this example the parameters to be considered are entries and

Copyright © 2011 John Wiley & Sons, Ltd.
DOI: 10.1002/spe

Mathematical Model: *Iteration Time (T_{Ex})*

Parameters

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

Mathematical Model

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

Figure 8. Mathematical model of the iteration time calculation.

exits of functions. They allow for the calculation of necessary time intervals. As we provide the performance model and explain all the required elements, we put both steps of the methodology (providing and understanding the model) together. This is why the second step of the methodology (i.e. the understanding of the model) is not elaborated as a separated section.

4.2. Third step: Interpreting the performance model in the application

Although the proposed mathematical model was developed *ad hoc* for this simple application, we have to determine the required measure points. In other words, in order to calculate each one of the components of T_{Comp} and T_{Comm} we have to determine the points in the application code that allow for the necessary calculations. Considering Figure 7, we represent the instants of the entries and exits of different functions as i_x. According to Sections 3.1 and 3.2, *events* are entities which allow for capturing timestamps and additional associated information if needed. We define the corresponding events to capture the timestamps of entries and exits (the bold text represents the name of each event):

- T_{Ini} can be obtained from the substraction between the exit of the *DivideData()* function and the entry of the *Initialization()* function (in the example of Figure 7, $T_{Ini} = i_3 - i_1$):
  - **Event_Div_Dat_Exit**: *actor*: p1; *place*: exit of Divide_Data(); *attr*: timestamp;
  - **Event_Init_Entry**: *actor*: p1; *place*: entry of Initialize(); *attr*: timestamp;
- T_{Fin} similar to the previous case, for T_{Fin} we consider *Finalize()* action (in the example, $T_{Fin} = i_{10} - i_9$):
  - **Event_Fin_Exit**: *actor*: p1; *place*: exit of Finalize(); *attr*: timestamp;
  - **Event_Fin_Entry**: *actor*: p1; *place*: entry of Finalize(); *attr*: timestamp;
- T_{Proc} is calculated by considering the time the first process starts the processing of its data (in the example, the entry of *Process()* for P_1) and the time the last process finishes the processing of its data (the exit of *Process()* for P_2) (in the example, $T_{Proc} = i_7 - i_4$). The order in which the data are processed cannot be ensured, and hence we have to instrument the Process() function of P_1 and P_2. Once the events are captured, some comparisons have to be carried out to determine the order of execution:
  - **Event_Proc_Exit**: *actor*: p1,p2; *place*: exit of Process(); *attr*: timestamp;
  - **Event_Proc_Entry**: *actor*: p1,p2; *place*: entry of Process(); *attr*: timestamp;

Copyright © 2011 John Wiley & Sons, Ltd.

- T_{Send} is the time spent sending messages, whenever this time is not overlapped with the computing time (in the example, $T_{Send} = (i_4 - i_3) + (i_8 - i_7)$):
  - **Event_Send_Exit**: *actor*: p1,p2; *place*: exit of Send(); *attr*: timestamp;
  - **Event_Send_Entry**: *actor*: p1,p2; *place*: entry of Send(); *attr*: timestamp;
- T_{Recv} is the time spent receiving messages (in the example, $T_{Recv} = i_9 - i_8$):
  - **Event_Recv_Exit**: *actor*: p1,p2; *place*: exit of Receive(); *attr*: timestamp;
  - **Event_Recv_Entry**: *actor*: p1,p2; *place*: entry of Receive(); *attr*: timestamp;

  Note that in the example, interval $(i_5 - i_4)$ is not considered as part of T_{Recv} because it is overlapped to T_{Proc}.

Then, by capturing these events, we can obtain the timestamp of the entries and exits of different functions as the code is executed, which will allow us to calculate the total time of an iteration.

4.3. Fourth step: Identifying the actors in the application

In this example, from Figures 7 and 6 we can distinguish the existence of two different kinds of actors: Process1 and Process2. Process1 executes a bit of computing, but is mainly devoted to manage and coordinate all the work, while Process2 is devoted to process the received data and send back the results. Clearly, these processes have a different functionality, which turns them into different actors. It is important to identify them as separated actors, as each one of them need to be instrumented in a different way according to the required measure points. For each one of the actors we have to determine some properties and associate the collected data as the corresponding attributes:

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

The property *class* is *none*, as this simple example is written using C language. If it had been written in C++ we would have needed the name of the class. At this point of the methodology, the user must formalize all this information using the Tunlet Specification Language, to automatically generate the tunlet. Finally, he/she will use our generator to automatically create the tunlet code.

4.4. Formalizing the specification

In this section we formalize the specification of some of the entities previously determined. The specification obeys the syntax of the Tunlet Specification Language presented in Section 3.3. For simplicity, we only concentrate on one actor: the *p1*, presented in Figure 9(a). For this actor we formalize the *identifier*, the *minimal* and *maximal* amount of them along the iteration (in this simple example only one), the *completion condition* (in this case determined by the end of the Finalize() method), and finally the *class* and the *executable file* where the actor is stored. Following that, we enumerate the attributes of the actor. One of the attributes is *TInit*, which is initialized in 0.0 (inic:TInit=0.0;) without dependencies (depinic:none), and whose value is determined by the timestamp of Event_Init_Entry. This explains the value of the attribute *dependency*, which during the translation step will indicate that the value of TInit has to be assigned when the tunlet handles the event Event_Init_Entry. Given that there is only an actor of kind *p1*, the information is stored in the record of p1[0]. Let us now consider the specification of *Event_Init_Entry*. In Figure 9(c) we highlight that the event is captured in the code of actor p1, and that it indicates the starting of a new iteration (controliter=begin). The event Event_Init_Entry has an associated attribute to indicate which iteration has just started. This attribute takes the value from the variable called *i* whose specification is provided in Figure 9(b). The event Event_Init_Entry together with the event *Event_Div_Dat_Exit* illustrated in Figure 9(d) (in particular their corresponding timestamps) allows

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

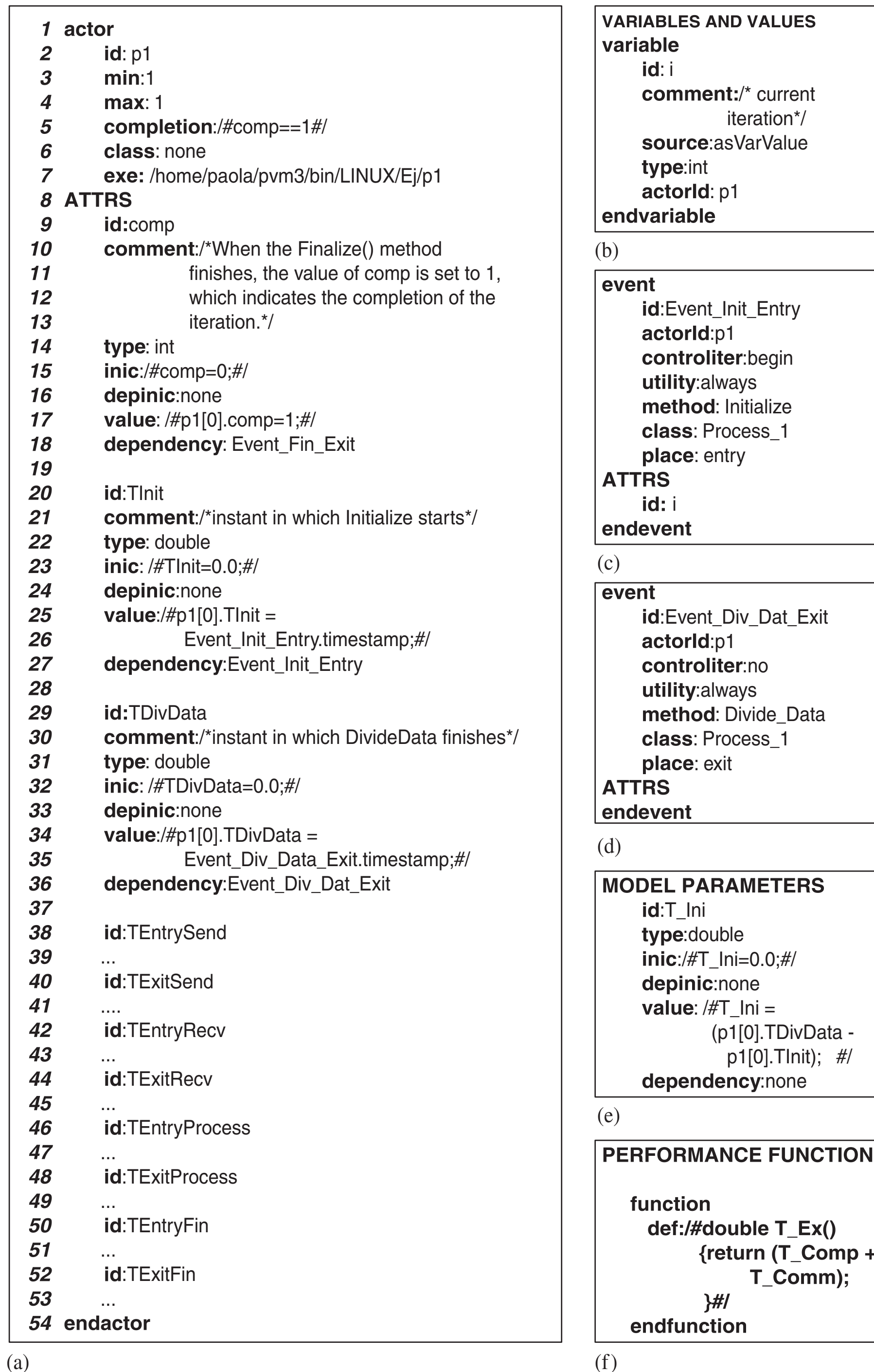

Figure 9. Extracts of the specification: (a) actor specification; (b) variable specification; (c) event specification; (d) event specification; (e) parameter specification; and (f) performance function specification.

for calculating *T_Ini*, the model parameter specified in Figure 9(e). The value of T_Ini is calculated according to the values of TInit and TDivData, two attributes of actor p1. Finally, in Figure 9(f), we formalize the main expression of the model, which allows for calculating the execution time.

These extracts of specification release the user from being directly involved with the specific implementation of MATE. Considering this simple example and this reduced set of entities specified

Copyright © 2011 John Wiley & Sons, Ltd.

```
class p1_Data
{
 public:
  p1_Data();
  void Set_comp(int c);
  void Set_TInit(double t);
  void Set_TDivDat(double t);
          ...
  double Get_comp();
  double Get_TInit();
  double Get_Set_TDivDat();
          ...
  bool IsComplete ();

 private:
  int comp ;
  double TInit ;
  double TDivDat ;
  double TEntrySend;
          ...
  double TExitFin;
};
```

(a)

```
bool p1_Data::IsComplete ()
{
 return (comp==1);
}

void p1_Data::Set_comp(int c)
{
 comp=c;
}

int p1_Data::Get_comp()
{
 return comp;
}
```

(b)

```
enum possibleEvents
{
 Event_Init_Entry = 1,
 Event_Div_Dat_Exit = 2,
 ...
 Event_Recv_Entry= 10
};
```

(c)

Figure 10. Extracts of source code related to the specifications—Declaration of data and events: (a) data of p1; (b) some methods of p1_Data; and (c) names of the events to capture.

```
void SimpleExampleTunlet::Insert_p1_Events (Task & t)
{
 // EVENT_INIT_ENTRY
 // Attribute List for Event_Init_Entry
 Attribute InitEntryAttrs [1];
 IterationStartsAttrs [0].source=asVarValue;
 IterationStartsAttrs [0].type=avtInteger;
 IterationStartsAttrs [0].id="i";
 Event InitEntry (Event_Init_Entry, "Process_1::Initialize", ipFuncEntry);
 InitEntry.SetAttribute (1, InitEntryAttrs);
 InitEntry.SetEventHandler(*this);
 t.AddEvent(InitEntry);

 // EVENT_DIV_DAT_EXIT
 // Attribute List for Event_Div_Dat_Exit
 Event DivDatExit (Event_Div_Dat_Exit, "Process_1::Divide_Data", ipFuncExit);
 DivDatExit.SetEventHandler(*this);
 t.AddEvent(DivDataExit);

 ...
}
```

Figure 11. Extracts of source code related to the specifications—Instrumentation of the application.

in Figure 9, the user is exempted from programming the corresponding source code, parts of which are presented in Figures 10–13. In the first place, the attributes of the actor p1 and the corresponding methods to set and get them have to be declared (Figure 10(a)). In Figure 10(b) we present the implementation of some Set/Get methods. In second place, an enumeration of the possible events has to be provided. In this case, *Event_Init_Entry* and *Event_Div_Dat_Exit* constitute the first and second elements of the enumeration possibleEvents (Figure 10(c)). Additionally, the events have to be created, registered with the EventHandler process (in general the tunlet constitutes the EventHandler) to determine which process will receive and process it. This is illustrated in

Copyright © 2011 John Wiley & Sons, Ltd.

```
void SimpleExampleTunlet::HandleEvent (EventRecord const & r)
{
 IterData * data = FindIterData (_lastIterIdx);
 switch(r.GetEventId ())
 {
  case Event_Init_Entry :
  {
        data->Get_p1(0).Set_TInit(r.GetTimestamp());
        _lastIterIdx=r.GetAttributeValue(0).GetIntValue();
        break;
  }
  case Event_Div_Dat_Exit :
  {
   data->Get_p1(0).Set_TDivDat(r.GetTimestamp());
   break;
  }
  default:
  {
        break;
  }
 }//end switch
}
```

Figure 12. Extracts of source code related to the specifications—Management of the received events.

```
void SimpleExampleTunlet::Update_T_Ini (double t)
{
 IterData *data =FindIterData(iterIdx);
 T_Ini=data->(Get_p1(0).Get_TDivData()) - (Get_p1(0).Get_TInit());
}

int  SimpleExampleTunlet::T_Ex( )
{
 return (T_Comp + T_Comm);
}
```

Figure 13. Extracts of source code related to the specifications—Treatment of model parameters and functions.

Figure 11, where we summarize the two events considered in this section. Note that in the case of the Event_Init_Entry, the variable *i* is associated as an attribute. In the third place, the EventHandler has to be provided with the functionality to process each kind of events, as shown in Figure 12. Finally, Figure 13 shows how to calculate T_Ini and T_Ex.

5. EXPERIMENTAL RESULTS

In the previous section we presented a simple and easy to understand example, which does not depend on any application implementation, but is useful to explain the complete methodology. In this section we briefly present some results of usefulness and effectiveness of MATE to automatically tune parallel applications using the proposed methodology. The transparent creation of tunlets from specifications relieves the programmer from being involved in the implementing details of MATE and the management of the collected information. To conduct our experiments, we considered a performance problem presented by Master/Worker applications: the amount of workers according to the computational work and the load balancing of the application. We focused on the performance model presented in Section 2.1. These are critical problems especially in time-sharing or heterogeneous environments, as the number of workers should be adapted depending on

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table I. Execution times of the application by itself and under MATE, in ms.

Experiment	Execution time
Application—1 worker	64 487
Application—2 workers	34 611
Application—4 workers	18 087
Application—8 workers	10 372
Application—16 workers	11 834
App. under MATE (tunlet created using DTAPI)	10 923
App. under MATE (tunlet created from specification)	10 887

the changes in the system. These experiments were conducted using an NBody application. The execution environment was a homogeneous cluster of Pentium 4, 1.8 GHz connected by 100 Mb/s network, with SuSE Linux 8.0 as operating system.

First, we only executed the application without any change and without the intervention of MATE. We measured the execution times when the application was executed in 1, 2, 4, 8 and 16 machines. Second, we executed the application under MATE in two different scenarios: in the first place, we implemented the model of the optimal number of workers [20] using DTAPI of MATE to create a corresponding tunlet; in the second place, we implemented the same performance model, but in this case we followed the proposed methodology to specify and create the tunlet.

In both scenarios the execution started with a unique worker, and then the number of workers was tuned according to the load in the system. The load pattern in the system was variable, but controlled and was the same for each experiment. We can divide the analysis of the results into two parts: benefits of using MATE for application tuning and effectiveness of the automatically generated tunlet. Considering the first part of the experimentation, in a general view of the results presented in Table I, we can observe the benefits obtained when the application is executed under MATE. The execution times under MATE are comparable to the best time obtained when the fixed number of workers is 8. However, when the application is executed under MATE, the computational resources are used only when they are really necessary, making a more efficient use of the whole system. The second part of the experimentation, in a particular comparison of the two last experiments, we can appreciate the effectiveness of the tunlet created from the specification. We can see that there is no additional overhead introduced into the application execution comparing both tunlets—the tunlet developed by hand directly using DTAPI and the specified and automatically generated one. As the proposed methodology is devoted to facilitate the use of MATE to improve the execution performance rather than to improve the code performance, this is a very important result given that the user obtains the same benefit in tuning the application working at a more abstract level. The complete reasoning to create this tunlet may be referred in [28].

6. CONCLUSIONS

In many scientific fields, parallel/distributed computing provides the necessary power to solve problems faster. In this work we have treated a very important aspect of high-performance computing: the transparent tuning process of parallel applications. In particular, we have focused on MATE which provides dynamic and automatic tuning of parallel applications, based on performance models implemented as tunlets. 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. In this work we have proposed and developed a particular extension of MATE in order to expand, improve, and facilitate its usability. We have transformed the MATE environment into more transparent tool for the users. We have given the user the possibility of improving the application performance in a simple way without the typical efforts related to the C/C++ implementation.

We have proposed and developed a methodology to automatically generate tunlets from specifications. The user only defines a set of abstractions taking into account information about the

Copyright © 2011 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

application to interpret the performance model under consideration. Some of these abstractions are 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, and name. These abstractions have to be formalized to constitute the specification, i.e. they must be written using the Tunlet Specification Language. In addition, we have developed and coded a translator to transform a given specification into a tunlet code. The whole methodology considers the requirements of MATE, i.e. the API that tunlets have to follow to work within MATE. Our approach constitutes a very promising way to extend the use of MATE, so that users can specify or reuse 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. Furthermore, we have shown the effectiveness of MATE to tune applications and the comparability of the results between the manual and the automatically generated version of the tunlet. Transparency of MATE is a quality necessary to make MATE a more useful and user-friendly tool. The proposal and developments presented in this work attempt to provide MATE with such characteristics. Although several improvements remain to extend and improve MATE, we established the basis to provide users the possibility of using MATE in a transparent way for the tuning process of parallel applications.

ACKNOWLEDGEMENTS

This research has been supported by the MICINN-Spain under contract TIN2007-64974 and by the CONICET-Argentina under contract PIP11220090100709.

REFERENCES

1. Grama A, Gupta A, Karypis G, Kumar V. *Introduction to Parallel Computing* (2nd edn). Pearson Addison Wesley: U.S.A., 2003.
2. Maillet E. *TAPE/PVM an Efficient Performance Monitor for PVM Applications—User Guide*. LMC-IMAG: Grenoble, France, 1995.
3. Geist A, Heath TM, Peyton BW, Worley PH. *A User's Guide to PICL: A Portable Instrumentation Communication Library. TR TM-11616*, Oak Ridge National Lab, 1990.
4. Heath M, Etheridge J. Visualizing the performance of parallel programs. *IEEE Software* 1995; **8**(5):29–39. DOI: 10.1109/52.84214.
5. Nagel W, Arnold A, Weber M, Hoppe H. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer* 1996; **12**:69–80.
6. Espinosa A, Margalef T, Luque E. Automatic detection of parallel program performance problems. *VECPAR'98* (*Lecture Notes in Computer Science*, vol. 1573). Springer: Berlin, 1998; 365–377. DOI: http://doi.acm.org/10.1145/281035.281051.
7. Yan J, Sarukhai S. Analyzing parallel program performance using normalized performance indices and trace transformation techniques. *Parallel Computing* 1996; **22**:1215–1237. DOI: 10.1016/S0167-8191(96)00032-4.
8. Miller B, Callaghan M, Cargille J, Hollingsworth J, Irvin R, Karavanic K, Kunchithapadam K, Newhall T. The paradyn parallel performance measurement tool. *IEEE Computer* 1995; **28**:37–46. DOI: 10.1109/2.471178.
9. Ribler R, Vetter J, Simitci H, Reed D. Autopilot: Adaptive control of distributed applications. *Proceedings of the Seventh IEEE Symposium on High Performance Distributed Computing*, Chicago, 1998; 172–179. DOI: 10.1109/HPDC.1998.709970.
10. Tapus C, Chung I, Hollingsworth J. Active harmony: Towards automated performance tuning. *Proceedings of the Conference on High Performance Networking and Computing*, Baltimore, 2002; 1–11. DOI: 10.1109/SC.2002.10062.
11. Mayes K, Lujan M, Riley G, Chin J, Coveney P, Gurd J. Towards performance control on the grid. *Philosophical Transactions of the Royal Society of London Series A* 2005; **363**/**1833**:1793–1806. DOI: 10.1098/rsta.2005.1607.
12. 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* 2007; **19**(11):1517–1531. DOI: 10.1002/cpe.v19:11.
13. Caymes-Scutari P, Morajko A, Margalef T, Luque E. Automatic generation of dynamic tuning techniques. *Euro-Par'07* (*Lecture Notes in Computer Science*, vol. 4641). Springer: Berlin, 2007; 13–22. DOI: 10.1007/978-3-540-74466-5_3.
14. Caymes-Scutari P, Morajko A, Margalef T, Luque E. Scalable dynamic monitoring, Analysis and tuning environment for parallel applications. *Journal of Parallel and Distributed Computing* 2010; **70**(4):330–337. DOI: 10.1016/j.jpdc.2009.08.008.

Copyright © 2011 John Wiley & Sons, Ltd.

15. 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 and Technology* 2005; **5**(3):115–120.
16. Morajko A, César E, Caymes-Scutari P, Margalef T, Sorribes J, Luque E. Automatic tuning of Master/Worker applications. *Euro-Par'05* (*Lecture Notes in Computer Science*, vol. 3648). Springer: Berlin, 2005; 95–103. DOI: 10.1007/11549468_14.
17. Mesa JG. Framework Master/Worker. *Master Thesis*. Universitat Autònoma de Barcelona, 2004; 1–154.
18. Buck B, Hollingsworth J. An API for runtime code patching. *The International Journal of High Performance Computing Applications* 2000; **14**:317–329. DOI: 10.1177/109434200001400404.
19. Mattson T, Sanders B, Massingill B. *Patterns for Parallel Programming*. Addison-Wesley: Reading, MA, 2004.
20. César E, Mesa J, Sorribes J, Luque E. Modeling Master-Worker applications in POETRIES. *Proceedings of the IEEE Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Santa Fe, NM, U.S.A., 2004; 22–30. DOI: 10.1109/HIPS.2004.1299187.
21. César E, Moreno A, Sorribes J, Luque E. Modeling Master/Worker applications for automatic performance tuning. *Parallel Computing* 2006; **32**(7):568–589. DOI: 10.1016/j.parco.2006.06.005.
22. César E, Sorribes J, Luque E. Modeling pipeline applications in POETRIES. *Euro-Par'05* (*Lecture Notes in Computer Science*, vol. 3648). Springer: Berlin, 2005; 83–92. DOI: 10.1007/11549468_12.
23. Extensible Markup Language (XML). Available at: http://www.w3.org/XML/ [September 2005].
24. Flex, a fast scanner generator. Available at: http://www.gnu.org/software/flex/manual/ [September 2005].
25. Document Object Model (DOM). Available at: http://www.w3.org/DOM/ [September 2005].
26. XSL Transformations (XSLT)—Version 1.0. Available at: http://www.w3.org/1999/XSLT/Transform [October 2005].
27. XQuery 1.0, XPath 2.0, and XSLT 2.0 Functions and Operators. Available at: http://www.w3.org/2005/04/xpath-functions [October 2005].
28. Caymes-Scutari P. Extending the usability of a dynamic tuning environment. *PhD Thesis*, TEX—Universitat Autònoma de Barcelona, 2007; 1–235.

Copyright © 2011 John Wiley & Sons, Ltd.