Automatic Generation of Dynamic Tuning Techniques*

Paola Caymes-Scutari, Anna Morajko, Tomàs Margalef, and Emilio Luque

Departament d'Arquitectura de Computadors i Sistemes Operatius, E.T.S.E, Universitat Autònoma de Barcelona, 08193-Bellaterra (Barcelona) Spain

Abstract. The use of parallel/distributed programming increases as it enables high performance computing. However, to cover the expectations of high performance, a high degree of expertise is required. Fortunately, in general, every parallel application follows a particular programming scheme, such as Master/Worker, Pipeline, etc. By studying the bottlenecks of these schemes, the performance problems they present can be mathematically modelled. In this paper we present a performance problem specification language to automate the development of tuning techniques, called "tunlets". Tunlets can be incorporated into MATE (Monitoring, Analysis and Tuning Environment) which dynamically adapts the applications to the current conditions of the execution environment. In summary, each tunlet provides an automatic way to monitor, analyze and tune the application according to its mathematical model.

1 Introduction

Nowadays, parallel/distributed applications are used in many science and engineering fields. They may be data intensive and may perform complex algorithms. Their main goal is to solve problems as fast as possible. Performance is a crucial issue on parallel/distributed programming. When a programmer develops an application, he/she expects to reach certain performance indexes. Therefore, it is necessary to carry out a performance analysis and tuning phase to fulfill the expectations. However, there are many applications that depend on the input data set or even can vary their behaviour during one particular execution according to the data evolution. In such cases, it is not worthy to carry out a postmortem analysis and tuning, since the conclusions based on one execution could be wrong for a new one. It is necessary to carry out a dynamic and automatic tuning of the application during its execution without stopping, recompiling nor rerunning it. In this context, the MATE environment was developed.

MATE (Monitoring, Analysis and Tuning Environment) [1,2] provides dynamic automatic tuning of parallel/distributed applications. The process followed by MATE to steer applications is showed in figure 1. During runtime, MATE automatically instruments a running application to gather information about its behaviour. The analysis phase receives events, which hold the collected information. Then, the performance functions are evaluated using that

* This work has been supported by the MCyT under contract TIN2004-03388.

information, in order to detect bottlenecks in the execution. According to the result of the evaluation, a solution is determined to improve the behaviour of the application. Finally, the application is dynamically tuned by applying the given solution. To modify the application execution on the fly, MATE uses the technique called *dynamic instrumentation* [3].

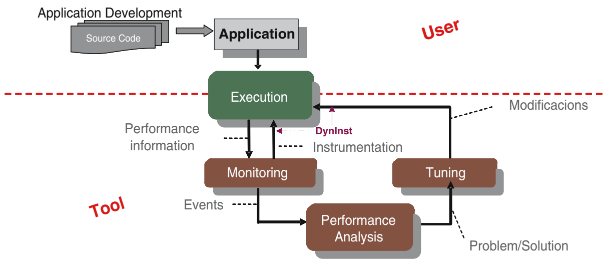

Fig. 1. Operation of MATE

All the information on how to solve a specific performance problem, is encapsulated as a piece of software called "**tunlet**". When MATE is executed, it loads a collection of tunlets in order to incorporate the knowledge to adapt the applications. Each tunlet implements a performance model and is used through the execution of the application to conduct the monitoring, analysis and tuning processes. Each tunlet condenses the description of a particular performance problem. The knowledge is represented by using the following terms: *measure points*, i.e. the locations in a process where the instrumentation must be inserted, such as a variable value or a function call; *performance functions*, i.e. activating conditions and/or formulas that model the application; *tuning points/actions*, i.e. the application components that must be changed to improve the performance.

In MATE, each performance problem should be separately tackled in a particular tunlet. This has as a consequence the need of implementing a tuning component for each performance problem. Until now, if users want to add a new tunlet to solve a problem not included previously, they should study the code of MATE to correctly implement their tunlet. Each tunlet is a C/C++ shared library that must be implemented using the Dynamic Tuning API provided by MATE (DTAPI). This added more complexity and effort to the programming and tuning tasks.

The goal of this work, is to develop a tool to automatically generate tunlets from specifications. The measure points, performance functions, tuning actions, and information about the application declared in the specification, will be used by the generator to automatically create the structures to allow the straightforward insertion of the new tunlet into the tuning environment.

So, in this paper we present a mechanism to automatically generate dynamic tuning techniques. In Section 2, we describe the performance problem specification language proposed to automatically create tunlets. In Section 3, we present a use case of the language: we provide an overview of the performance model to tune the number of workers in a Master/Worker application, and show the deduced specification needed to generate the tunlet. Section 4 shows some experimental results obtained by applying the automatically generated tunlet. Finally, Section 5 summarizes the conclusions of this work.

2 Automatic Generation of Tunlets

In this section we describe the whole tool which was designed to automatically generate tunlets. Notice that this is a general tool and can be used for any parallel application, whenever the user has enough knowledge about it to define the parts of the specification. Thus, the user has to define a set of abstractions in function of the application and the performance model to specify the tunlet. Such abstractions makes the user think in the tunlet as MATE does. In consequence, this tool makes easier and more transparent the usage of MATE. In the following, we firstly present the specification language, and secondly, the automatic generation process is described.

2.1 Tunlet Specification Language

When a language is defined, it is needed to analyze and consider what must be included, i.e. the elements, the relationships among them, its syntax and semantics [4,5]. In the particular case of specifying tunlets, it is needed to examine and consider the elements of the performance model and of the application, having in mind how the Analyzer represents and uses the knowledge, and the DTAPI that tunlets should use to correctly work in MATE.

From the **performance model** point of view, it is needed to consider the *measure points*, the *performance functions*, and the *tuning actions and points*, owing to they provide the metrics and the means to evaluate and adapt the behaviour of the application. In addition, it is necessary to determine the variables, functions, etc. in the specific application needed to interpret the model, i.e. to stablish a correlation between performance model parameters and entities in the application, to be able to collect the necessary information. Thus, from the point of view of the **application** we need to be aware of the *programming model* it follows, i.e. how the different kinds of processes or *actors* are involved in the scheme; the *variables or values* we can manipulate, both to get their value or to change them, and the functions whose execution we need to collect the information and send it as *events*.

Therefore, the specification of a tunlet is divided into three different sections, which we describe in the following paragraphs. The grammar of the tunlet specification language has been defined, but it is not included in this paper due to legibility and space reasons.

With regard to **measure points**, it constitutes the larger part of the specification. The user must define:

– The **actors** of the programming model: the types of processes co-existing in parallel, such as master and worker in the Master/Worker model, or each one of the phases in the Pipeline model. 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. These three elements are needed to obey the DTAPI requirements. Some additional information is required:
  - the minimal and maximal quantity of this type of actor could co-execute is needed to generate the structures to manage the behaviour information of each process along the succesives iterations.
  - 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; for example for a worker, to catch the computation time along the iteration could be interesting.
- The **events** to capture, such as entries or exits of functions. Each event is defined by its name, the actor it is asociated with, the place in the source code and a certain key number to indicate if the event controls the beginning or the end of the iteration. 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 metric caught when an event that indicates the exit of the sending function occurs. Each event has two default attributes: *tid* and *timestamp*, to indicate the task it was caught in and the specific instant in which it happened.
- The **variables**, i.e. the entities in the program whose value can vary. They can be instrumented or tuned in the application. For each one must be declared: name, data type, if in the program it is a variable, a parameter or a function output, and the actor who has visibility of it. In general, these entities are used in determining the value of the attributes of the remaining specification elements.
- The **iteration information**, includes an attribute to indicate the current iteration, and, according to the performance model, all the additional information necessary to describe the behaviour of each iteration, such as total communication or processing time in the iteration.
- The **model parameters** are the attributes of the performance model, whose value generally is calculated as a function of the attributes of actors or events.
- 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, the name of such entity should be expressed.

Regarding to the performance functions, they are 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.

Finally, for each tuning point it must be declared the name, the way in which its value must be calculated, a condition to apply the tuning, and some information about synchronization: the appropriate place and instant to change the value of the point.

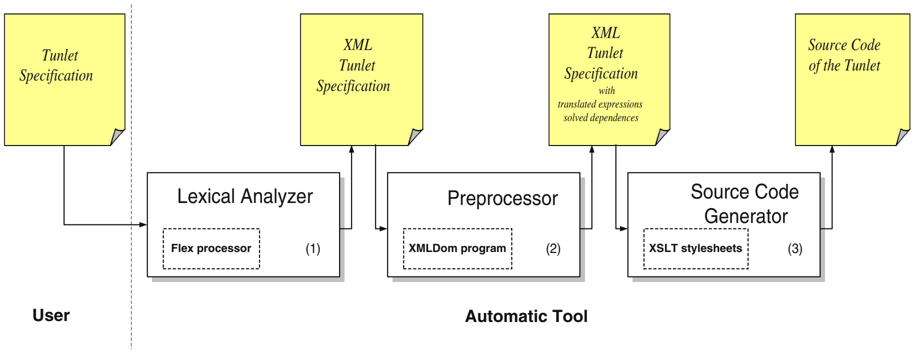

Fig. 2. Generation of a tunlet from a specification

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**) attributes (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 selecting the right attribute such as in any data structure by using the dot (*actor[i].attribute*). Information asociated with events and iteration information are accessed in a similar way (*event.attribute, iter.attribute*).

2.2 Tunlets Generation

Once the user defined the specification, the translation process to obtain the source code of the tunlet follows several steps, shown in Figure 2:

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 [6]. This step exists for user-friendliness reassons, and consists in translating the specification into its equivalent XML specification. The lexical analyzer was implemented in Flex [7].
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 [8]
3. *Source Code Generation*: the input is the XML specification obtained in preprocess phase. The output are 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 [9,10].

3 Use Case

In this section we include an example on how to specify a tunlet from a given performance model. The model we present is the optimal number of workers for a Master/Worker programming scheme [11]. Taking into account this model, we will specify the tunlet to tune a Master/Worker application to solve the NBody problem.

3.1 Optimal Number of Workers Model

One of the major performance problems in Master/Worker applications is related to the quantity of workers used to process the tasks. The performance model we are using deals with homogeneous applications, and it is assumed there are only one process for each processor.

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 depends on the following parameters, which should be captured as indicated:

- 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).

This expression is 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 [11]).

3.2 Number of Workers Tunlet Specification

In this section we define the specification of a tunlet to tune the number of workers by considering the above-mentioned performance model. In the following, we analyze how to specify each one of the involved entities. Notice that given the

```
actors
        master
                min: 1
                max: 1
                completion:/#true#/
                class:_CMaster
                exe: /users/paola/pvm3/bin/LINUX/master
            attributes
                firstSend
                    comment:/*first send of the master*/
                    type: double
                    inic:/#firstSend=0.0;#/
                    value: /#  if (MSendsTaskW.timestamp<master[0].firstSend
                                    || master[0].firstSend==0.0 )
                                  { master[0].firstSend= MSendsTaskW.timestamp;} #/
                    dependency: MSendsTaskW
                lastRecv
                lastWorker
        worker
```

(a)

```
events
        IterationStarts
            actor:master
            controliter:3
            method: _M_SendIteration
            class:_CMaster
            place: ipFuncEntry
          attributes
            iteration
            NumTuples
            WorkUnit
        IterationFinishes
        MSendTaskW
        WRepliesM
        WStartsT
        WFinishesT
```

(b)

Fig. 3. (a) Actors in the application. (b) Events to catch during the execution.

length of the entire specification, in each subsection we only show one element in details (written in ***Bold and Italic*** fonts), even the rest of the elements are mentioned too. To start with the *measure points* section, we declare two kinds of actors: *master* and *worker*. As shows the figure 3*(a)*, the Master process is defined in the *_CMaster* and *_MyMaster* classes, and its executable is called "*master*". The *min* and *max* properties indicate that only one instance of the master can exist along the execution. The interesting attributes for the master are the time in which the first task in the iteration is sent (*firstSend*), the time in which the last answer is received (*lastRecv*) and the identifier of the worker that finished the processing at last (*lastWorker*). These three attributes are necessary to calculate communication time, needed in the expression used to obtain λ. In the case of the *firstSend* attribute, it is a *double* variable, whose initial value is 0.0. This value changes when an incoming *MSendsTaskW* event has a *timestamp* earlier than it, or that is the first *MSendsTaskW* event received. This fact clearly indicates that the value of *firstSend* depends on the reception of the MSendsTaskW events.

In the *events* section there are the events to determine the beginning and ending of an iteration (*IterationStarts* and *IterationFinishes*), the communication time (*MSendTaskW* and *WRepliesM*), and the computing time (*WStartsT* and *WFinishesT*). All of them are shown in the figure 3*(b)*. In particular, we describe the *IterationStarts* specification. This event has to be caught when *master* executes the *_M_SendIteration* method of *_CMaster* class, particularly at the entry of the method, which is indicated by the *place* property (*ipFuncEntry*). The attributes that the event should record, are *iteration*, *NumTuples* and *WorkUnit*, which are needed to register the current iteration, the data volume to being processed along an iteration, and the size of each data unit. These three attributes defined in the application, are specified in the *variables* section. The last two of them are useful to calculate the value of v_i.

The *variables* which could be "manipulated" in the application in order to dynamically obtain their value or change them, are *Iteration* (variable used to store the current iteration), *NumTuples* (data volume to process), *WorkUnit* (size of each data unit), *ResSize* (size of the reply a worker send to the master), *workerTID* (identifier of the worker), *nw* (number of workers currently used) and *NoptWorkers* (optimal number of worker to be used). See the figure 4*(a)*. In particular, *Iteration* is an integer variable (*asVarValue*) which is in the scope of the master.

```
variables
        iteration
           comment: /*current iteration of the Master process*/
           source: asVarValue
           type: int
           actor:master
        NumTuples
        WorkUnit
        ResSize
        workerTID
        nw
        NoptWorkers
```

(a)

```
iteration information
        StartedIteration
           comment: /*last iteration registered */
           type: int
           inic: /#StartedIteration=0;#/
           value: /#iter.StartedIteration=IterationStarts.iteration;#/
           dependency:IterationStarts
        totalwork
        tuplesize
        iterCommTime
        availableMachines
```

(b)

Fig. 4. (a) Susceptible of change or reading variables. (b) Iteration information.

In the case of the *iteration information* section, it includes the default attribute -*StartedIteration*- and some additional attributes needed to the particular performance model: *totalwork*, *tuplesize* and *iterCommTime*. As shows the figure 4*(b) StartedIteration* has to be initialized in 0, and its value should be updated to *iteration* each time an *IterationStarts* event happens.

This is all about the entities the users have to determine and define in order to obtain the values of each parameter of the model necessary to evaluate the performance functions. In figure 5*(a)* we enumerate the performance parameters of the model. In particular, v_i (or *vi*) is an integer whose value depends on *IterationStart* event, due to it carries the value of *NumTuples* and *WorkUnit*, needed to calculate the size of tasks sent to each worker, as explained in section 3.1. As can be seen in figure 5*(b)*, the specification includes the performance function to calculate the optimal number of workers ($pf()$), which is the implementation

```
performance model parameters
        vi
           comment: /*data volume  to be processed*/
           type:int
           inic: /#vi=0;#/
           value: /#vi=NumTuples*WorkUnit;#/
           dependency: IterationStarts
        vm
        V
        Tc
        n
        tl
        lambda
```

(a)

```
Performance Functions
     def:/#int pf(){ int nro=0; nro=(int)sqrt( (lambda * V + Tc)/tl);
             return nro;}#/


Tuning Points
     NOptWorkers
         value:/#pf()#/
         syncfunction:0
         syncplace:0
         cond:/#NOptWorkers<availableMachines #/
```

(b)

Fig. 5. (a) Performance parameters. (b) Performance function and Tuning points.

of the expression presented in Section 3.1. When *pf()* is evaluated, the result is assigned to the tuning point, *NoptWorkers*. Depending on the condition, in this case if the currently available resources are enough, the tuning is effected instantaneously, due to the *syncfunction* and *syncplace* do not present particular requirements.

4 Experiments

In this section we want to validate the usefulness of the tunlet automatically generated for the example presented in Section 3. We compare the execution time of the application when executed under the automatically generated tunlet, under the hand made tunlet and when executed by itself in different fixed number of workers (1, 2, 4, 8, 16, 19). To conduct the experiments, we selected a 2D N-Body created by using the Master/Worker framework [12]. Experiments were conducted on a cluster of homogeneous Pentium 4, 1.8 Ghz, (SuSE Linux 8.0) connected by 100Mb/sec network. We created certain load patterns, so that we can introduce and modify certain external loads to simulate the systems timesharing. Each experiment was performed many times and the average of the execution time for the application was calculated and is shown in Table 1. In the last two scenarios, the application started with one worker and then, during the execution, the number of workers has been dynamically changed according to the model described in Section 3.1. As can be seen, the automatic development of the tunlet from the specification by using our proposed tool, results in a tunlet capable of tuning the application in a time comparable to the hand made tunlet. This means that the performance of MATE is not degraded by the use of tunlets automatically generated.

Table 1. Execution time (in seconds) of NBody when executed in different scenarios

Fixed number of workers	1	2	4	8	16	19
Execution Time	64,49	34,61	18,09	10,37	11,83	15,49
NBody + handmade tunlet	Starting with 1 worker and then tuning					
Execution Time	10,92					
NBody + generated tunlet	Starting with 1 worker and then tuning					
Execution Time	10,89					

5 Conclusion

The increasing use of high performance computing and the necessity of improving the use of the resources, reflects the need of tools to assist the user in tuning his/her parallel applications. We presented a tunlet specification language. Each tunlet is defined from the performance model of the problem under consideration and information about the application and the parallel programming scheme it follows. Specifications are automatically translated into source code to be included in MATE. In this way, if a user knows a specific performance problem in the application and knows the mathematical model to overcome the problem, he or she can develop a specification of it which will automatically be translated into a tunlet to use in MATE in order to tune the program during runtime. This allows for extending MATE and facilitates its usage, due to the user does not need to know any MATE implementation details. In consequence, we make easier the use of dynamic and automatic tuning of parallel applications.

References

1. Morajko, A., Morajko, O., Jorba, J., Margalef, T., Luque, E.: Dynamic Performance Tuning of Distributed Programming Libraries. In: Sloot, P.M.A., Abramson, D., Bogdanov, A.V., Gorbachev, Y.E., Dongarra, J.J., Zomaya, A.Y. (eds.) ICCS 2003. LNCS, vol. 2660, pp. 191–200. Springer, Heidelberg (2003)
2. Morajko, A., Morajko, O., Margalef, T., Luque, E.: MATE: Dynamic Performance Tuning Environment. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 98–107. Springer, Heidelberg (2004)
3. Buck, B., Hollingsworth, J.K.: An API for Runtime Code Patching, University of Maryland, Computer Science Department. Journal of High Performance Computing Applications (2000)
4. Aho, A., Ullman, J.: The Theory of Parsing, Translation, and Compiling, vol. 1. Prentice Hall, Englewood Cliffs (1972)
5. Aho, A., Sethi, R., Ullman, J.: Compilers - Principles, Techniques, and Tools. Addison-Wesley Publishing Company, London, UK (1986)
6. Extensible Markup Language (XML), `http://www.w3.org/XML/`
7. Paxon, V.: Flex, a fast scanner generator (1995), `http://www.gnu.org/software/flex/manual/`
8. Document Object Model (DOM), `http://www.w3.org/DOM/`
9. XSL Transformations (XSLT) - Version 1.0, `http://www.w3.org/TR/xslt`
10. XQuery 1.0, XPath 2.0, and XSLT 2.0 Functions and Operators, `http://www.w3.org/2005/04/xpath-functions`
11. César, E., Mesa, J.G., Sorribes, J., Luque, E.: Modeling Master-Worker Applications in POETRIES. In: IEEE 9th International Workshop HIPS 2004, IPDPS, April, 2004, pp. 22–30 (2004)
12. Mesa, J.G.: Framework Master/Worker, Universitat Autònoma de Barcelona, Departament d'Informàtica. Master (2004)