Monitoring, Analysis and Tuning Environment: Classification, Intrusion and Overhead*

Paola Caymes-Scutari[1], Anna Morajko[2], Tomàs Margalef[2] and Emilio Luque[2]

[1] Departamento de Ingeniería en Sistemas de Información, Universidad Tecnológica Nacional - Facultad Regional Mendoza. (M5502AJE) Mendoza, Argentina.
[2] Departament d'Arquitectura de Computadors i Sistemes Operatius, Universitat Autònoma de Barcelona, 08193-Bellaterra (Barcelona) Espanya

Abstract. The increasing use of high performance computing has been motivated by the requirements of the scientific applications –such as the data set size or the complexity of the operation– and empowered by the advances in the abilities of the processors and the interconnection networks technology. However, the parallel programming paradigm involves additional aspects to the merely functional which could provoke different kinds of bottlenecks in the performance of the applications. Since the performance is a key issue specially in grand challenge problems, the applications must be optimized or tuned to use the computational resources in an efficient way. Different approaches and tools are available to assist the users in the tuning process at different levels. In this paper we present an overview about the performance tuning problematic, the advantages and disadvantages of the performance tools and analyze the particular case of MATE (Monitoring, Analysis and Tuning Environment), which is a dynamic and automatic tuning tool specially suitable for time' sharing or heterogeneous systems. The analysis is focused on the intrusion and the overhead caused by the environment over the own performance of the application, and demonstrate that the obtained benefits are much more significant than the overhead provoked.

1 Introduction

In the last years, the computing performance demand has been in increase. Many applications involve a big data set or very complex operations such as the determining of the human genome, the simulation of the universe, study of nature models, etc. These classes of applications require the use of systems of great computer power. Due to the fact that the improvement of the speed of operation of the sequential systems (the processors and other components) is restricted by the speed of the light, the thermodynamic laws, and the high financial costs for the manufacture of the processor, the scientific comunity has been directing the attention towards the parallel/distributed paradigm. A viable and cost-effective solution is to connect multiple processors together and to coordinate their computational power. The resulting systems are popularly known

* This work has been supported by the MCyT under contract TIN2007-64974.

as *parallel computers*, and they allow for the sharing of a computational task among multiple processors [2]. The general and basic idea is that n processors or nodes should provide a computational speed n times faster than a simple node, i.e., the problem should be solved in an interval of $1/n$ of the time [4]. Clearly, the advantages of using parallel systems constitute an ideal situation which in practice is not always true. However, even though parallel systems have some limitations in execution time, those limits are upper than the uniprocessors ones.

When using parallel systems, the development of parallel applications has to follow a specific manner to allow for their execution in a parallel system. In addition, once the application has been implemented, it has to be systematically tested from the functional point of view in order to guarantee its correctness. Following that, the application has to be adjusted to ensure that no bottlenecks exist in the execution, and in consequence that fulfils the aim of providing a better performance. In consequence, given that the performance of the parallel applications is a key aspect, the programmers have to face a series of difficulties to reach the best performance of their applications. Through the years, several indices have been defined in order to evaluate the deployment of parallel computing. Some of the performance parameters, such as the *execution time*, the *scalability*, the *efficiency* and the *load balance*, have a general importance. Nevertheless, none of those indices provides the users with specific information or suggestions to overcome the problems. Then, the development of an application with a good performance, forces the users to face the optimization process, so-called *tuning process*.

The tuning process includes several and successive phases to adapt and improve the behaviour of the applications by modifying their critical parameters. Firstly, during a *monitoring phase* the information about the behaviour of the application is captured. Next, the information is *analyzed*, by looking for bottlenecks, deducing their causes and trying to determine the adequate actions to eliminate them. Finally, the appropriate changes have to be applied to the code to solve the problems and improve the performance. As a consequence, the developers are forced to know very well the application, the different involved software layers and the behaviour of the distributed system. All these issues make difficult and costly the performance tuning process, specially for non-expert users, due to a high degree of expertise is required in order to significantly improve the behaviour of the application. Fortunately, through the years different approaches and tools have been developed with the aim of helping the user during some of the optimization phases (monitoring, analysis or tuning phases). In the following section we depict the main goals, characteristics, advantages and drawbacks of the different approaches to tune applications. In Sect. 3 we document MATE [9, 3] , a dynamic and automatic tuning tool based on performance models. In Sect. 4 we present the study carried out about the overhead and intrusion caused by MATE at the different levels of its operation. Finally, in Sect. 5 we present the main conclusions of this work.

2 Performance Approaches

The basic purpose of the performance tools is to help a programmer to understand the performance characteristics of an application. In particular, the tool should analyze and locate the parts of the application that exhibit poor performance and cause program bottlenecks. Such tools are categorized as *monitoring tools*, *analysis tools* or *tuning tools*.

Most performance monitoring tools consist of some or all of the following components [2]:

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

With regard to the performance analysis tools, since their objective is to automate the evaluation of the monitored information, they include some performance knowledge and some mechanism to find bottlenecks and provide solutions. The complexity and the philosophy of the analysis process determines how fast the solutions or modifications are available to be introduced into the application. In the case of the performance tuning tools, they include some mechanism to instrument the code with the aim of automating the process of inserting modifications into the application to overcome the detected bottlenecks. Some tools cover more than one of these categories, helping the user in more than a simple level. The usage of these performance tools offers several benefits and drawbacks. On the one hand, the tools alleviate the responsibility and knowledge of the user about parallelism and performance problems. On the other hand, the performance tools introduce some overhead on the normal operation of the program. A particular issue is the intrusiveness of the tracing calls and their impact on the applications performance. It is very important to note that instrumentation affects the performance characteristics of the parallel application and thus provides a false view of its performance behaviour [2]. However, the overhead is in general hidden by the benefits.

Every tool shares the goal of helping users to tune the behaviour of their applications. Through the years, several approaches in performance monitoring, analysis and tuning have been proposed in order to assist the users in improving their applications. In the following subsections we provide an overview of them. In particular, since our research is focussed on the dynamic tuning approach, we mention some of the most important tools in the approach and we compare them with MATE, the core of our work.

2.1 Classical Performance Analysis

The classical performance analysis approach is based on the *post-mortem* analysis of the application behaviour carried out by the user. Figure 1 presents the

general flow of this kind of analysis. First, while the application is running, a monitoring tool obtains information about the behaviour of the application. The corresponding instrumentation had been statically inserted by the monitoring tool or manually by the user. When the application is being executed and performing the instrumented code, the instrumentation allows for data measurements and collection. The collected information is stored in a trace file. Once the execution of the application finishes, the trace file is used to interpret and understand the behaviour of the application. Then, in the following step, the performance data are graphically interpreted by some visualization tool. During the performance analysis phase, the graphics are useful to help the user to understand the tracing in order to analyze the behaviour presented by the application through the execution. In the last step, the user manually changes the source code of the application in accordance with decisions made during the analysis. Then, the modified program has to be re-compiled and re-linked for future executions. This process is successively repeated until an acceptable performance is achieved.

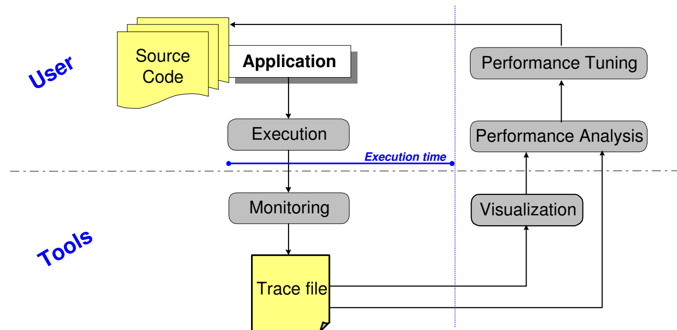

Fig. 1. Classical performance analysis approach

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

2.2 Automatic Performance Analysis

The automatic performance analysis approach releases the user from having a high degree of expertise in parallel systems and performance analysis. This is shown in Fig. 2. When the execution of the application finishes, the analysis tool looks for performance bottlenecks automatically by considering the information collected by the monitoring tool and its own knowledge about potential problems the application can present. Depending on the philosophy of each tool, the performance knowledge is represented and managed in a particular way (heuristics, history, fuzzy logic, etc.). The knowledge allow for detection of bottlenecks as well as their causes and needed changes to improve the future executions of the application. When the performance analysis finishes, it provides the user with the corresponding suggestions to modify the source code. As in the previous approach, the user changes the application, re-compiles and re-links it to the next execution.

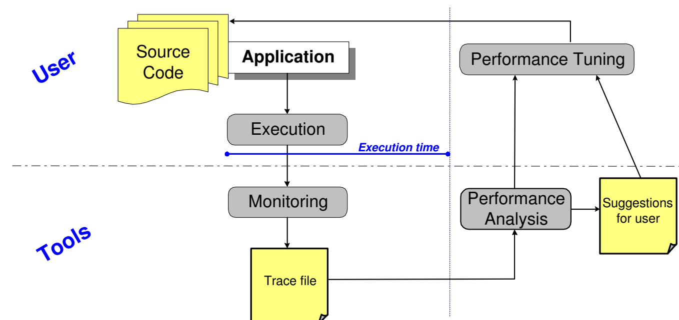

Fig. 2. Automatic performance analysis approach

Although this approach exempts the user from the very difficult and time consuming task of analysing the behaviour of the application, it has some constraints. On the one hand it is still based on trace files and considers a single execution of the application; on the other hand, the creation of knowledge models is not an easy task and need a trade-off between simplicity and accuracy. Then, it is only suitable for the same set of applications as in the classical approach. Some examples of tools following this approach are **KappaPi** [5] and **Paradise** [6]

2.3 Dynamic Performance Analysis

The dynamic performance analysis proposes to overcome the drawbacks presented by the *post-mortem* analysis, such as the analysis based on a single run of the application and on large trace files. The analysis is made "*on the fly*" by considering performance data collected by an *on-line monitoring tool*, which

presents the benefit of independence from a trace file. Figure 3 shows the general view of this approach. The instrumentation can be dynamically inserted into or eliminated from the application by applying dynamic instrumentation techniques [1].

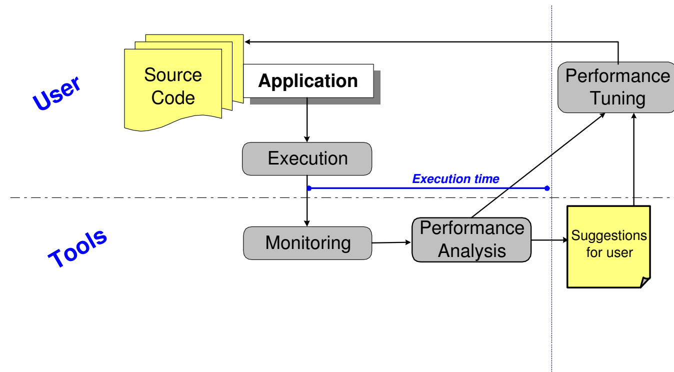

Fig. 3. Dynamic performance analysis approach

The dynamic analysis approach allows for detection of performance problems faster than the *post-mortem* approaches. It is suitable for iterative long-running applications. However, it requires the user to stop, modify, recompile and re-run the application in order to apply the tuning. Then, as in the previous approaches, decisions based on a single execution could not be significant in future executions, when the application depends on the input data, their evolution, or the state of the system. **Paradyn** [8] is an example tool in the dynamic performance analysis approach.

2.4 Dynamic Performance Tuning

The previous three approaches have been incrementally overcoming the difficulties presented by their precedent approaches. The dynamic performance tuning approach offers automatic tuning during run-time instead of manual insertion of changes in the source code. Figure 4 shows the general operation of this approach.

All the phases in this approach are carried out during run-time. The analysis step is not based on trace files, but it uses the measurements provided by a dynamic monitoring tool. According to the evaluation of the performance, the tuning actions are automatically and dynamically carried out in the application. Thus, the running parallel application would be automatically monitored, analyzed and tuned on the fly without need to re-compile, re-link nor restart it. This completely exempts the users from taking part in tuning their applications. Another advantage is that the performance of the application is evaluated

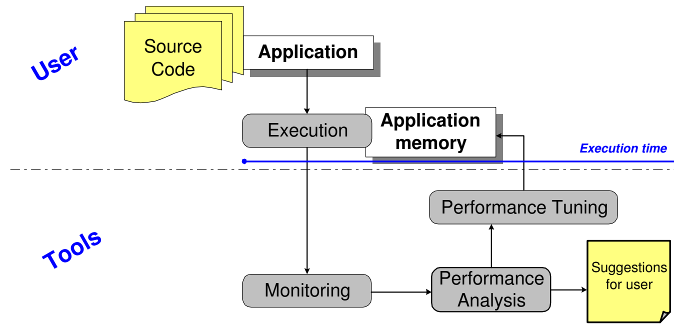

Fig. 4. Dynamic performance tuning

and tuned according to its current behaviour in the environment. Then, the decisions are more accurate and consistent, due to every execution of the application is separately tuned according to its particular execution conditions, i.e. different input data or different conditions in the execution environment. Such as in the previous approach, this one is suitable for iterative, long running and resource-intensive programs. In particular, this approach is specially suitable for applications runing in a time sharing or heterogeneous environment.

There exist different tools in this approach, such as Autopilot [10], Active Harmony [11], PerCo [7], and MATE [9, 3]. Due to MATE is the core of this work, we dedicate an overview of it in Sect. 3, and mention what makes it different from the others tools.

Autopilot, automatically chooses and configures resource management algorithms based on application request patterns and observed system performance. It provides a set of *performance sensors*, *decision procedures* and *policy actuators*. Autopilot relies on fuzzy sets and uses a set of IF-THEN production rules that map the sensor input values to the actuator output space.

Active Harmony, permits automatic adaptation of algorithms, data distribution, and load balancing. It integrates different libraries with the same or similar functionality. The user's application uses such a set of libraries with different algorithms and tunable parameters to be changed. During runtime Active Harmony monitors underlying library execution and manages the values of the different parameters. The system is able to select a more efficient library and change tunable parameters to improve the application performance.

PerCo (*Performance Control*), can be used for distributed applications executing on a heterogeneous network, such as a computational Grid. PerCo is capable of monitoring the progress of the applications and redeploying them so as to optimize performance. PERCO requires performance prediction capabilities, such as history of previous executions.

3 MATE

MATE is an environment which provides dynamic and automatic tuning for parallel/distributed applications. The tuning of the application comprises three different phases: monitoring of the behaviour of the application, performance analysis of the collected information and tuning of the application. All these phases are continuously and automatically executed on the fly. The knowledge about what to measure, how to evaluate the behaviour and what to change to adjust the behaviour is based on mathematical performance models, specified by the user in terms of the applications [3]. The main goal of MATE is to improve the performance of an application, by adapting it to the variable current conditions of the system. Hence, the user is exempted from manual application tuning. MATE is composed by several components which cooperate among them to control and to improve the execution of the application. The main components are the following:

(i) **Application Controller** (*AC*): it is a daemon like process which controls the execution and the dynamic instrumentation of the individual tasks. AC is composed of the *Monitor* and *Tuner* modules that cooperate to provide the required functionality:
   - **Monitor** is the module responsible for instrumenting and monitoring the execution of an application. The monitoring is based on function calls event tracing.
   - **Tuner** is the module responsible for applying the tuning actions over the application tasks. The needed changes are determined by the solutions proposed by the Analyzer.

(ii) **Dynamic Monitoring Library** (*DMLib*): this is a shared library which is dynamically loaded in the application tasks. It is used to perform the data monitoring and collection.

(iii) **Analyzer**: this process carries out the performance analysis of the application. Internally, the Analyzer operates through *tunlets* which are one of most important components:
   - **Tunlets** constitute the core of dynamic and automatic tuning implemented by MATE, in terms of representation of knowledge. Each tunlet defines and implements a particular tuning technique, i.e. the logic to overcome a particular performance problem by encapsulating the knowledge about the performance problem in the following terms:
     (i) **Measure Points**, which indicate *what* is needed to measure in the application to be able of evaluating its behaviour. This definition includes values of variables, parameters, function returning, timestamps, etc.
     (ii) **Performance Functions**, those are mathematical expressions that determine *how* to evaluate the collected information in order to detect bottlenecks.
     (iii) **Tuning Points/Actions** indicating *what*, *where* and *when* to introduce the changes in the application execution with the aim of adapting its behaviour.

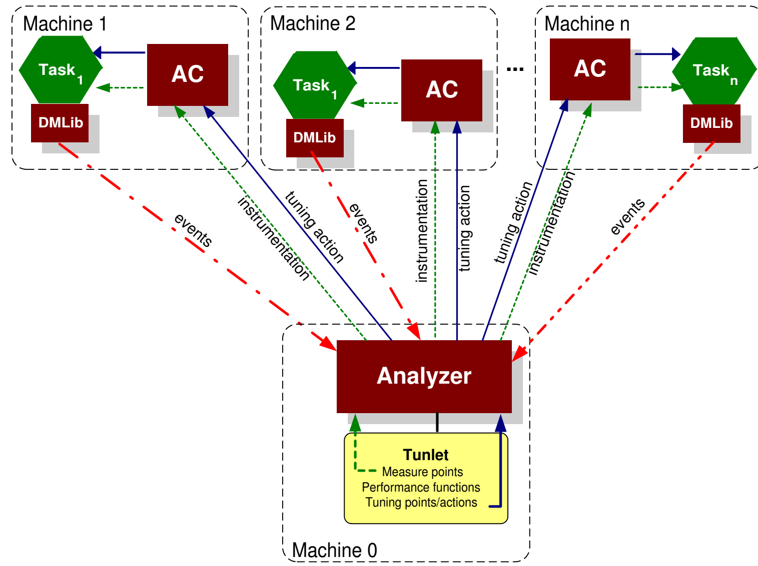

Fig. 5. Analyzer interacting with the rest of the environment

Figure 5 illustrates the execution of an application under MATE and the function of each module of MATE. When the execution of the application under MATE starts, a particular tunlet indicates to the Analyzer what the set of measure points required is. Analyzer forwards the requirement to every AC (distributed over every task). In Fig. 5, these requirements of instrumentation are represented by the dashed arrows. Then, through the execution the Analyzer receives requested event records generated by different processes and the tunlet is notified (the dashed-dotted arrows in the figure). The events are classified according to their type (i.e., among the different events to be caught -such as start of an iteration, entry to a certain function, etc.- what kind of event it is) and the associated information stored in their attributes is used in order to update the values of parameters in the performance model. The update is carried out (*directly* when some event attribute embodies the value of a performance parameter or *indirectly* when the event attribute has to be temporally stored until all the information necessary to evaluate the performance parameter is available) When all the information of the iteration is received, extracted (from the events) and processed, the tunlet evaluates the performance functions to determine the current and optimal performance. If the tunlet detects a performance bottleneck, it decides if the current performance can be improved in existing conditions. If so, the tunlet informs the Analyzer about the possible improvement. In consequence, 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, and hence to the Tuner (illustrated by continuous arrows). Using MATE, the run-time changes of the application, for both the

monitoring and tuning processes, are implemented via the dynamic instrumentation library DynInst [1]. Different experiments demonstrated the usefulness and benefits of using MATE. More details about MATE, its approach and the provided benefits can be consulted in [9, 3]. In this paper we analyze the intrusion provoked by MATE.

Even though MATE shares some characteristics with the tools presented in Sect. 2.4, it has some particularities. On the one hand, if we consider the preparing of the application to be tuned, using MATE the monitoring is based on the dynamic instrumentation where the application does not require to be prepared for tuning due to measure and tuning points are inserted on the fly. In Autopilot the developer must prepare the application inserting sensors and actuators manually into the source code. In Active Harmony the mechanism is based on the integration of different libraries with the same functionality. On the other hand, if we consider the way in which the performance analysis is carried out, MATE uses simple, conventional rules and performance models, whilst Autopilot uses fuzzy logic to automate the decision-making process, Active Harmony uses heuristic algorithms in order to describe the application behaviour, and PerCo is based on history.

4 Overhead caused by MATE

The use of some tool to supervise and/or improve the application execution has some inherent advantages and drawbacks. As mentioned in Sect. 2, the instrumentation and the monitoring processes affect the performance characteristics of the parallel application providing a false or altered view of its own performance. In the particular case of MATE, the tuning process constitutes another source of overhead for the application. In consequence, it is very important to analyze not only the benefits and the improvements obtained in the application performance when using MATE, but it is necessary to consider the overhead caused by the use of MATE to adapt the behaviour of the application. In this section we document the study of the aspects related to the intrusion of MATE. To perform the analysis we divided the intrusion in three different types, depending on the nature of the intrusion: caused by the instrumentation, the monitoring or the tuning process. In the next paragraphs, we first depict each kind of source of intrusion. Secondly, we document the experiments carried out to measure the intrussion and finally we analyze the results obtained.

4.1 Sources of Intrusion

Instrumentation. The instrumentation allows for inserting new code in the application at certain points, with the aim of collecting information when the execution of the application passes by such places. In the particular case of the dynamic instrumentation, it allows also for removal of inserted code. Depending on the moment in which the instrumentation is inserted (or removed), we can consider two different cases:

– *Initial instrumentation*: in general, the bulk of the instrumentation is inserted when the application begins its execution, to start the collection of data as soon as possible. Thus, when the application starts, it enters in a phase of overhead caused by the instrumenting process, and just then continues or resumes the execution.
– *Extra instrumentation*: as mentioned before, some performance models requires the addition (or removal) of some instrumentation, due to additional information is necessary (or unnecessary) according to the current conditions of the system and the behaviour of the application.

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

Monitoring. The monitoring process consists in detecting the instrumented points in the application to collect the required information. As depicted in Sect. 3, the monitoring process is based on function calls event tracing. The monitoring process is implemented by means of the cooperation of the Monitor module of AC's and the DMLib. The Monitor supports the management of the set of monitored events according to the requirements of the Analyzer, whilst the DMLib facilitates the data collection and is responsible for registration of events. During the instrumentation process, the Monitor creates a piece of code (called "*snippet*") which is inserted in the application when the registering of a new event is required. In this way, when a snippet is invoked during the execution all the attributes associated to the event are obtained, and the event is registered via DMLib. Each event includes at least *timestamp, event identifier, number of attributes,* and *attributes.* DMLib uses a set of buffers in order to minimize the network overhead when sending events. The sending of events is controlled by time to avoid excessive wait for individual events. The communication with Analyzer is established by using an event collection low level protocol based on TCP/IP.

Tuning. The tuning process introduces changes in the application. This could be at level of variables or at level of functions. The overhead provoked will fundamentally depend on the kind of tuning required. The Tuner is the module responsible for applying the tuning actions over the application tasks. The needed changes are determined by the solutions proposed by the Analyzer. The Tuner modifies the application execution via DynInst, by modifying the memory associated to the application. The Tuner provides an API which defines the set of tuning actions that the Analyzer can require:

– *LoadLibrary*: this loads a certain library in a process. This allows Analyzer to load additional code required to the tuning process.

- *SetVariableValue*: the value of a certain variable in a determined process can be modified.
- *ReplaceFunction*: this allows for replacing every call to a certain function for a call to another function.
- *InsertFunctionCall*: a new function call with its attributes can be inserted.
- *OneTimeFunctionCall*: it allows for calling a certain function once during the execution.
- *RemoveFunctionCall*: every call to a certain function is eliminated.
- *FunctionParamChange*: the value of a parameter can be changed in the entry of a function, before the body of the function is being executed.

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

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

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

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

In this case, there is no coherence among the value of *nw* considered at the different points of the iteration. Such incoherences provoke an abnormal or unexpected behaviour in the application, whose consequences could be disastrous. In this particular example, the master process becomes blocked waiting for the answers which never will be received.

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

- *by synchronizing the modifications*: in this case, the change of the value of *nw* can be made exactly before the next iteration starts. Then, a breakpoint should be inserted to stop the execution in such a point, then the value is changed and the execution is resumed. In this way, the value of *nw* will be established through the iteration.
- *by using an auxiliary variable*: this is perhaps the more pragmatic approach to introduce modifications, due to it does not require an additional stopping and resuming of the execution to insert a breakpoint. However, in this case the cooperation of the user is required in case a new variable has to be added in the application. In the example, we could use an auxiliary variable *nworkers* as follows:

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

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

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

4.2 Experimental Measurements

In this section we present the results obtained from the experiments carried out to measure the overhead provoked by each kind of source of intrusion or overhead. To conduct the experiments, we selected a Master/Worker application, given that such parallel algorithm model is widely used to implement different models in the scientific field thanks to its widely spread use and flexibility. In particular, we have used a 2D N-Body implementation, which calculates the new position of N particles in each iteration, according to the mutual interaction among them. The experiments were conducted on a homogeneous cluster composed by Processors PENTIUM IV 3.0 Ghz, 1 GB DDR-SDRAM 400 Mhz, Ethernet card Broadcom NetXtreme Gigabit and Fedora Core 4 as operating system. All the machines were configured to use NFS (Network File System) based on one server with the same characteristics as the cluster machines. Each measurement was repeated thirty times, and the average has been calculated. The time wasted to insert the *instrumentation* to catch an event is about 0,3 *ms*. In the case of the *monitoring*,

the capture and sending of an event takes about 0,8 *ms*. On its hand, the time wasted to execute some *tuning* action varies from 0,25 to 1700 *ms*, depending on the nature of the action. The time wasted when a breakpoint has to be inserted before applying the tuning action is 1,539 *seconds* in average. Table 1 presents in more details the results obtained from the experimentation, where the time is expressed in miliseconds and represents the average time for each case.

Table 1. Average Time wasted in the different tuning actions, in *ms*

SOURCE	KIND	TIME
Instrumentation	Initial	0,284
	Extra	0,284
Monitoring		0,844
Tuning	Set Variable Value	1,1858
	Replace Function	2,0078
	Insert Function Call	1,308
	Remove Function Call	0,254
	Function Parameter Change	0,4007
	On time function call	1678,286

4.3 Analysis of the Results

From the obtained results documented in 4.2, we can analyze the impact of each kind of source of intrusion along the complete execution of the application. In the analysis we analyze the results in terms of absolute overhead rather than in terms of percentages of improvement or overhead, since percentages are inherent to the total execution time of each particular application and execution environment configuration. In addition, along with the analysis we intend to define an expression to estimate the overhead caused by MATE, according to each particular application and performance model.

With respect to instrumentation, the time consumed in initial instrumentation could be considerable given that each performance model involve several measure points which are obtained by means of events (i.e. by capturing the corresponding values at the entry or exit of some function), and the instrumentation of each particular event takes about 300 microseconds). However, the overhead caused by the initial instrumentation is only suffered at the start-up of the application and is disguised as the execution of the applications progresses. Remind in general we are considering big applications, which could execute several minutes

or hours. Thus, the time taken by initialization (in order of miliseconds) is hidden by the total execution time (in order of minutes or hours). With respect to monitoring, note that the overhead is proportional to the amount of events to be caught and the number of times that each event occurs through the iteration. The monitoring actions are executed along with each iteration of the application, whenever the corresponding instrumentation is still in the application. Capturing each event takes about 850 microseconds. Therefore, we can consider that the monitoring process is which introduces a continuous overhead in the application execution, different from instrumenting and tuning (remind that the instrumenting process in general causes the major overhead just at the start-up of the application and eventually when some additional instrumentation has to be inserted or removed). Finally, with respect to the tuning process, it introduces overhead as the conditions of the environment change and some adaptation in the application is necessary; thus, if the conditions are not changing along every iteration not too many tuning actions will be required and in consequence not too much time will be wasted in tuning. The time wasted could vary among 250 microseconds and 1.5 seconds depending on the complexity of the tuning action.

Considering the previous analysis, we can estimate the intrusion caused by MATE using the following expression:

$$T_{Instr}(e_1...e_n) + T_{Mon}(e_1...e_n) + T_{Tun}(app) \ . \quad (1)$$

where $T_{Mon}(e_1..e_n) = \sum_{i=1}^{n} T_{Monitor}(e_i) * Occurr(e_i)$. T_{Instr} is the time wasted to insert the instrumentation in the application to catch the corresponding n events; T_{Mon} represents the time wasted to catch and send the events, and T_{Tun} represents the time used to tune the application.

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

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

5 Conclusions

Parallel systems provide the computational power required by applications that involve intensive calculations. However, the parallel paradigm presents some performance bottlenecks inherent to the benefits of using multiple resources. Different approaches and tools assist the users to solve and overcome the bottlenecks. Each tool provide the help in a different way. However, the supervision of the applications carried out by the performance tools introduces some overhead in

the execution. We studied the particular case of MATE, a Monitoring, Analysis and Tuning Environment which implements dynamic and automatic tuning based on mathematical performance models. From the experiments, we concluded that even though MATE provokes a certain overhead from the order of microseconds to miliseconds (against the long lasting HPC applications), the cost of such negative effects over the application is not significant in comparison to the benefits obtained along with the execution of the complete application, since the successive tuning decisions allow for a better and more efficient use of the involved resources. This makes MATE specially suitable for dynamic parallel applications. We also defined an expression to calculate a priori the overhead caused by MATE according to the measurements and tuning actions involved by the performance model under consideration.

References

1. Buck, B., Hollingsworth, J.: An API for Runtime Code Patching. University of Maryland, Computer Science Department. Journal of High Performance Computing Applications (2000)
2. Buyya, R. et al: High Performance Cluster Computing - Architectures and Systems (Volume 1). Prentice Hall (1999)
3. Caymes-Scutari, P., Morajko, A., Margalef, T., Luque, E.: Automatic Generation of Dynamic Tuning Techniques, LNCS 4641 Euro-Par 2007 Parallel Processing, 13–22 (2007)
4. Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., White, A.: Sourcebook of parallel computing. Morgan Kaufmann Publishers (2003)
5. Espinosa, A., Margalef, T., Luque, E.: Automatic Detection of Parallel Program Performance Problems. Lecture Notes in Computer Science, vol. 1573, pp. 365–377, Springer-Verlag (1998)
6. Krishnan, S., Kale, L. V.: Automating Parallel Runtime Optimizations Using Post-Mortem Analysis. International Conference on Supercomputing, 221–228 (1996)
7. 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, Vol. 363, No. 1833, 1793–1806 (2005)
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 vol. 28, 37–46 (1995)
9. 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, 19/11, 1517–1531 (2007)
10. Ribrel, R., Vetter, J., Simitci, H., Reed, D.: Autopilot: Adaptive Control of Distributed Applications High Performance Distributed Computing 1998, 172–179 (1998)
11. Tapus, C., Chung, I-H., Hollingsworth, J.: Active Harmony: Towards Automated Performance Tuning. High Performance Networking and Computing 2003, 1–11 (2003)