Dynamic Performance Tuning Environment [1]

Anna Morajko, Eduardo César, Tomàs Margalef, Joan Sorribes, and Emilio Luque

Computer Science Department. Universitat Autònoma de Barcelona
08193 Bellaterra, Spain
ania@aows10.uab.es;
{eduardo.cesar,tomas.margalef,joan.sorribes,emilio.luque}@uab.es

Abstract. Performance analysis and tuning of parallel/distributed applications are very difficult tasks for non-expert programmers. It is necessary to provide tools that automatically carry out these tasks. Many applications have a different behavior according to the input data set or even change their behavior dynamically during the execution. Therefore, it is necessary that the performance tuning can be done on the fly by modifying the application according to the particular conditions of the execution. A dynamic automatic performance tuning environment supported by dynamic instrumentation techniques is presented. The environment is completed by a pattern based application design tool that allows the user to concentrate on the design phase and facilitates on the fly overcoming of performance bottlenecks.

1 Introduction

Parallel and distributed processing are very promising approaches due to the high computation power offered to the users. However, designing and developing parallel/distributed applications are quite hard tasks because of the lack of tools that facilitate several aspects of parallel/distributed processing.

The programmer must take into account the low level implementation details of the parallel programming model being used. This fact implies that the programmer must be very careful to avoid dangerous situations such as deadlocks. When the first version of the application is completed, the programmer must consider the correct functioning of the application. It implies to go through a functional debugging phase to fix all bugs and ensure that the program provides the right functionality. The mentioned phases are also present in classical sequential programming, but the complexity of these tasks is significantly higher on parallel/distributed systems.

Once the program works correctly, the programmer must analyze the application performance. The decision of developing a parallel application is related to the performance requirements, and therefore the programmer must be aware of obtaining a satisfactory performance. This new task is not common in sequential programming because in most cases sequential programmers rely on compiler optimizations.

[1] This work was supported by the Comisión Interministerial de Ciencia y Tecnología (CICYT) under contract number TIC 98-0433.

This performance analysis is a hard task that requires using some performance analysis tools. The classical way to carry out performance analysis is to run the application using some monitoring tool that creates a trace file. This trace file includes all the events recorded during the execution of the application. The task of the programmer is to analyze these events looking for performance bottlenecks found during the execution in order to determine their causes and modify the program to improve the performance. There are several visualization tools that try to help the user by providing a set of different views of the program execution [1, 2]. The user can analyze the views searching for performance bottlenecks and this is much easier to do than reading plain trace files. The use of such visualization tools allows the user to detect the main performance bottlenecks quite easily, but the most difficult task is to relate the performance bottleneck to its causes. Therefore, these tools do not provide direct information that can guide the programmer in the performance improvement phase. Tuning a parallel/distributed application is a difficult task that requires great experience and deep knowledge about the program and the system itself.

There are some tools that go a step ahead providing not only visualization facilities, but offering some hints that try to guide the user in the decision that must be taken to improve the performance [3, 4]. The main feature of these tools is that they provide some automatic techniques that analyze the trace file and provide some hints to the non-expert user in the performance tuning phase.

However, the use of trace files and the fact that the analysis is done in a post-mortem phase imply some constraints in the applicability of the methodology:

1. The trace files can be really big. The execution of a real application that takes some hours usually creates enormous files that are difficult to manage and to analyze in order to find the performance bottlenecks.
2. The use of a single trace file to tune the application is not completely significant, specially when the application behavior depends on the input data set. In this case, the modifications done to overcome some bottleneck present in one run may be inadequate for the next run of the application. This could be solved by selecting a representative set of runs, but the provided suggestions would not be specific for a particular run.
3. Certain applications can change their behavior during their execution. In this case, the modification suggested by the post-mortem analysis cannot cover the dynamic behavior of the application.

In these conditions, the post-mortem methodology is not the best approach since the modifications that could be carried out from the suggestions provided by the performance analysis tool do not cover all the possible behaviors of the application. Therefore, the only feasible solution to provide real performance tuning facilities is to carry out the performance tuning on the fly.

Dynamic performance tuning is an important facility to be incorporated in a parallel/distributed programming environment. This approach is presented in section 2, showing the design of a dynamic performance tuning environment. Section 3 introduces a pattern based application design environment, that facilitates the program tuning on the fly. Section 4 shows an example of how an application is designed and tuned using our dynamic performance tuning environment. Finally, section 5 presents some conclusions of this work.

2 Dynamic Performance Tuning Environment

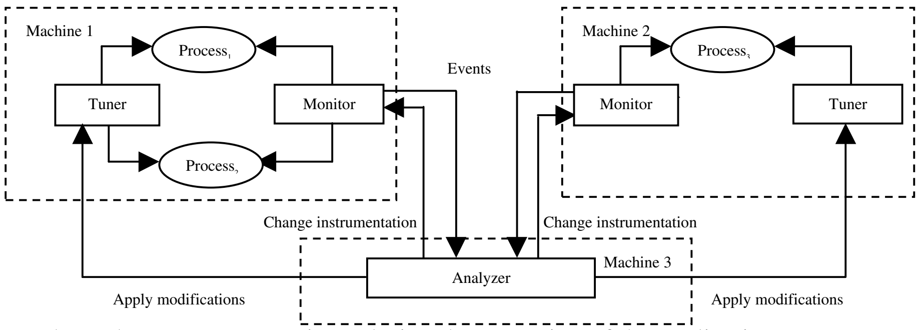
The main goal of our dynamic performance tuning environment is to improve performance by modifying the program during its execution without recompiling and rerunning it. This goal requires several steps: monitoring of the application, analyzing the performance behavior and modifying the running program to obtain better performance.

2.1 Dynamic Performance Tuning of Parallel/Distributed Applications

Since parallel applications consists of several intercommunicating processes executed on different machines, it is not enough to improve task performance separately without considering the global application view. To improve the performance of the entire application, we need to access global information about all associated tasks on all machines. The parallel application processes are executed physically on distinct machines and our performance tuning system must be able to control all of them. To achieve this goal, we need to distribute the modules of our dynamic tuning tool to machines where application processes are running. So, our dynamic performance tuning environment consists of several tools (see figure 1):

1. The monitoring tool (*monitor*) inserts the instrumentation in all the processes and collects the events produced during the execution of the application. However, our approach to dynamic analysis requires the global ordering of the events and thus we have to gather all events at a central location.
2. The analysis tool (*analyzer*) receives all the events generated in different processors during the execution, detects the performance bottlenecks, determines the causes and decides which modification should be done on the application program. The analysis may be time-consuming, and can significantly increase the application time execution if both – the analysis and the application - are running on the same machine. In order to reduce intrusion, the analysis should be executed on a dedicated and distinct machine. Obviously, during the analysis, *monitor* modules are collecting and providing new data to the *analyzer*. In some cases, the *analyzer* may need more information about program execution to determine the causes of a particular problem. Therefore, it can request the *monitor* to change the instrumentation dynamically. Consequently, the *monitor* must be able to modify program instrumentation – i.e., add more or remove redundant instrumentation – depending on its necessity to detect performance problems.
3. The automatic tuning tool (*tuner*) inserts modifications into the running application processes. When a problem has been detected and the solution has been given, *tuner* must apply modifications dynamically to the appropriate process or processes.

Using this dynamic methodology when a bottleneck is detected, the appropriate modifications are inserted into the address-space image of the running program. These changes will be invoked the next time the application reaches that point. The methodology can only be applied to problems that appear several times during the execution of the application. This fact could seem to be a constraint. However, it must be pointed out that the main performance problems of parallel/distributed application



are those that appear many times during the execution of the application.

Fig. 1. Dynamic performance tuning

This dynamic tuning approach is a real challenge that requires the use of some dynamic instrumentation technique that allows the inclusion of some new code in a running program without recompiling or even rerunning it.

2.2 Dynamic Instrumentation

The Paradyn group developed a special API called DynInst API [5] that supports dynamic instrumentation.

DynInst is an API for runtime code patching. It provides a C++ class library for machine independent program instrumentation during application execution. DynInst API allows the users to attach to an already running process or start a new process, create a new piece of code and finally insert created code into the running process. The next time the instrumented program executes the block of code that has been modified, the new code is executed. Moreover, the program being modified is able to continue its execution and does not need to be re-compiled, re-linked, or restarted. DynInst manipulates the address-space image of the running program and thus this library needs access only to a running program, not to its source code.

The DynInst API is based on the following abstractions:

- point – a location in a program where new code can be inserted, i.e. function entry, function exit
- snippet – a representation of a piece of executable code to be inserted into a program at a point; a snippet must be build as an AST (Abstract Syntax Tree).

Taking into account the possibilities offered by the DynInst library, it is possible to insert code in the running application for two main objectives:

- Insert code for monitoring purposes to collect information about the behavior of the application.
- Insert code for performance tuning. The main goal of the dynamic tuning is to improve the performance on the fly. Therefore, it is necessary to insert new code in the application.

Thus, the *monitor* and the *tuner* have been designed using the DynInst API.

2.3 Pattern Based Application Design for Performance Tuning

Dynamic performance tuning of parallel applications is a task that must be carried out during the application execution and therefore, there are some points to be considered:

1. It is necessary to introduce the smallest possible intrusion. Besides the classical monitoring intrusion in dynamic performance tuning there is some extra intrusion due to the monitors' communication, the performance analysis and the program modification.
2. The analysis must be quite simple because the decisions must be taken in a short time to be effective in the execution of the program.
3. The modifications must not involve a high degree of complexity, because it is not realistic to assume that any modification can be done on the fly.

For all these reasons, the analysis and modifications cannot be very complex. However, if the programmer can use any structure, the generated bottlenecks can be extremely complicated and the analysis and the modifications are extremely difficult.

A good compromise between performance improvement and application development is to offer the programmer a pattern based application design and development environment. Using this environment the dynamic performance tuning tool is simplified, because the set of performance bottlenecks to be analyzed and tuned are only those related to the programming patterns offered to the user.

On the other hand, this approach also facilitates application design and development. The user is constrained to use a set of programming patterns, but using them he/she skips the details related to the low level parallel programming. Moreover, the dynamic performance tuning environment (including the application design tool and the dynamic performance tuning tool) allows the programmer to concentrate on the application design and not to worry about the program performance.

3 Application Design Tool

It is well known that designing, developing and tuning parallel/distributed applications is a hard task. There is a lack of useful environments that facilitates these tasks. Our "pattern based" application design tool is useful to achieve two main goals:

- On the one hand, it facilitates modification of the application on the fly to improve the performance. The defined patterns are well-known structures that may present certain well-known performance bottlenecks. Therefore, the points that must be monitored to detect the possible bottlenecks as well as the actions that must be carried out to break them are well defined. Actually, each abstract pattern is characterized by a set of parameters that determines the dynamic behavior of the pattern. Thus, changes in the application behavior, to improve performance, could be carried out by modifying the values of these parameters.
- On the other hand, it simplifies the application design to the programmer offering a set of well-known useful programming structures. When one of the abstract

patterns has been selected by the programmer, the low-level details related to process communication are automatically created in such a way that the programmer must only fill in some methods corresponding to the particular details of the application being developed. This approach has been proposed many times in the literature as in [6, 7]. The main drawback of this approach is the lost of efficiency when compared with expert programming using low level primitives. However, our approach includes hidden performance improvement. Using our environment the programmer does not have to care about performance issues since the application will be tuned on the fly.

We have adopted a general methodology to characterize patterns. This methodology allows using a unified approach to the definition, analysis and implementation of each pattern, but also defines a way to include new patterns in the future. The methodology includes:

- General description of the pattern.
- Define the elements to specify the user application in terms of the selected pattern (interface). It includes initialization and ending, functional description, and communication management.
- Characterize the associated pattern bottlenecks.
- Determine the magnitudes that have to be calculated to detect these bottlenecks: which must be monitored.
- Determine the parameters that could be changed to overcome these bottlenecks and the actions that could be taken on them.

Using this application design tool the programmer is constrained to choose and combine the defined patterns. To ensure that the design tool is really useful it is necessary to include the most used programming patterns. Therefore, it is necessary to define a limited set of patterns that can be easily analyzed and tuned on the fly, but that also offer the possibility to design a wide range of parallel/distributed applications. The patterns included so far are the following ones:

- **Master/Worker**: *Description:* this pattern contains a master process that generates requests to other processes called workers, these requests are represented by data units called tasks. These workers make some computation on these tasks and then send the results back to the master. *Interface:* how master generates tasks (Task Generation), the actual computation that each worker must do (Task Processing) and the processing that the master must do on the received results (Results Processing). *Bottlenecks:* execution time differences among workers, too few workers, or too many workers. *Detection magnitudes:* communications time, workers computation time. *Adjustable parameters:* task distribution, and number of workers.
- **Pipeline**: *Description:* this pattern represents those algorithms that can be divided in an ordered chain of processes, where the output of a process is forwarded to the input of the next process in the chain. *Interface:* work that must be carried out at each stage of the pipe, input and the output data and connection among stages. *Bottlenecks:* significant executions time differences among stages, bad communication/computation ratio. *Detection magnitudes:* computing time

and data load of each stage, stage waiting time. *Adjustable parameters:* number of consecutive stages per node, number of instances of a stage.

- **SPMD (Single Program Multiple Data)**: *Description:* represents those algorithms where the same processing is applied on different data portions. *Interface:* specify the task that must be carried out for all the processes, the interconnection pattern (all-to-all, 2D mesh, 3Dcube, and so on) and the data that must be sent/received for each process. *Bottlenecks:* load imbalance, execution time differences among processes. *Detection magnitudes:* computing time and data load of each process, waiting time for other processes. *Adjustable parameters:* number of inter-communicating processes per node, number of processes, and, in some cases, data distribution.
- **Divide and Conquer:** *Description:* each node receives some data and decides to process it or to create some new processes with the same code and distribute the received data among them. The results generated at each level are gathered to the upper level. There each process receives partial results, makes some computation on them and passes the result to the upper level. *Interface:* processing of each node, the amount of data to be distributed and the initial configuration of nodes. *Bottlenecks:* important differences among branch completion time due to branch depth or process execution time. *Detection magnitudes:* completion time of each branch, computation time of each process, branch depth. *Adjustable parameters:* number of branches generated in each division, data distribution among branches, and branch depth.

Object oriented programming techniques are the natural way for implementing patterns, not only due to its power to encapsulate behavior, but also to offer a well defined interface to the user. For these reasons, we have designed a class hierarchy, which encapsulates the pattern behavior, and also a class to encapsulate the communication library. We use C++ language to implement our environment and we support PVM-based applications.

In next section we show our dynamic performance tuning environment, presenting the main aspects related to design, implementation, monitoring, performance analysis and tuning of a real application based on the Master/Worker pattern.

4 Case Study: Forest Fire Propagation

The forest fire propagation application [8] is an implementation of the "fireline propagation" simulation based on the Andre-Viegas model [9]. The model describes the fireline as a set of sections that can be desegregated to calculate the individual progress of each section for a time step. When the progress of all the sections have been calculated, it is necessary to aggregate the new positions to rebuild the fireline. The general algorithm of this application using the master/worker paradigm is:

Master:

1. Get the initial fireline
2. Generate a partition of the fireline and distribute it to the workers.
3. Wait for the workers answer.

4. **If** the simulation time has been finished **then** terminate
   else Compose the new fire line, adding points if needed and go to step 2.

Worker:

1. Get the fireline section sent by the master
2. Calculate the local propagation of each point in the section (to calculate the new position of a point the model needs to know its left and right neighbours).
3. Return the new section to the master.

Next we present the specification of the associated pattern, and then we describe the on the fly performance analysis and the results of the dynamic performance tuning of the application.

4.1 Pattern Specification

Once a pattern has been chosen for the application, the user must implement the interface elements of the pattern and define the data structures for tasks and results. Specification of the master-worker pattern associated with this application includes:

For the master.

Initialization: Load initial data (initial fireline).

Task Generation: Convert fireline into a set of tasks (groups of three contiguous points).
Send tasks.

Results Processing: Get results.
Merge results to calculate the new fireline (adding new points if needed).
If the simulation is completed indicate this fact.

Terminate: Show or save the final results (final fireline)

For the worker.

Task Processing: For each received task do

- Calculate new position for the central point (using the simulation model and information about neighbors)
- Generate result - the new point position

Data structures definition:

- Task structure: given that the simulation model is applied on groups of three contiguous points, a task must include three points.
- Result structure: for each group of three contiguous points the new position of the central point is calculated, then a result consists of a single point.

4.2 Monitoring and Performance Analysis

Once the application has been designed it is necessary to analyze the performance on the fly. The *monitor* inserts instrumentation to measure the time taken by each worker and the communication delays among the master and the workers (*detection magnitudes*). The instrumentation is inserted in those code segments where master sends and receives data, it is just after Task Generation and before Results Processing. In the worker, instrumentation is inserted before and after Task Processing.

In this application all the workers spend approximately the same amount of time (the application is executed on a homogeneous system). Therefore, in the first steps of the propagation simulation, the adequate number of workers is calculated according to the performance parameters of the pattern. However, when the fire propagates, new sections must be interpolated to accomplish the continuity constraint required by the propagation model. This means that as the fireline grows, each worker receives more data to proceed. After a certain number of iterations, the *analyzer* detects that the master is mostly waiting for results from the workers (*too few workers bottleneck*), because they must calculate a bigger amount of tasks increasing its computation time (*workers computation time magnitude*). When the *analyzer* detects this fact, it calculates the ideal number of workers for the new situation. In the case when this number is bigger from the current one, the *analyzer* decides to spawn a new worker (or several) (*increasing workers number parameter*).

4.3 Dynamic Performance Tuning

The *tuner* receives the decision from the *analyzer* to improve the performance of the application by spawning a new worker (or several) in a new machine. The *tuner* modifies the *adjustable parameter* "number of workers" of the master to let it know that there is a new worker (or several) and in the beginning of the next iteration the data must be distributed among all the workers, the old ones and the new ones. Using the dynamic performance tuning environment, the number of workers grows according to the requirements of the application.

In Table 1 the obtained results on an example fireline that starts with 50 points and reaches 850 points after 200 iterations are presented. For this example, the starting number of workers is 5 and it increases till 20. The table summarises the evolution of the number of workers and the execution times (in ms) for this particular example. In this table *Iter.* is the iteration step in the forest fire simulation, *No. points* is the number of points in the fire line for that particular iteration, *No. workers* is the number of workers required by the performance model. $T_{i\ min}$, $T_{i\ max}$ and $T_{i\ ideal}$ are the execution times for that particular iteration when executed with 5, 20 or ideal number of workers (determined by the *analyzer*), and $T_{t\ min}$, $T_{t\ max}$ and $T_{t\ ideal}$ are the total execution times when executed with 5, 20 or dynamic number of workers (ideal number for each iteration).

Table 1. Evolution of the number of workers and the execution times for forest fire example

Iter.	No. points	No. workers	$T_{i\ min}$	$T_{i\ max}$	$T_{i\ ideal}$	$T_{t\ min}$	$T_{t\ max}$	$T_{t\ ideal}$
1	50	5	11660	22902	11660	11660	22902	11660
50	250	11	34300	30512	26635	1171960	1362057	1017880
100	450	15	56940	38122	36880	3464260	3081712	2621468
200	850	20	102220	53342	53342	11444860	7662522	7171129

It can be observed that the dynamic creation of new workers provides the best execution time. It must be considered that, with dynamic tuning, resources are supplied only when needed. Using the right number of workers at each stage is even better than spawning the maximum number from the beginning. In the initial iterations a low number of workers provide better results than a bigger number because of the computation/communication relationship (compare T_i's for a low number of iterations). In the last iterations a higher number of workers provides better results. The overall execution times show that the dynamic creation of workers provides the shorter execution time.

5 Conclusions

A new approach to parallel/distributed performance analysis and tuning has been presented. The developed environment includes a pattern based application design tool and a dynamic performance tuning tool. The sets of patterns included in the pattern based application design tool have been selected to cover a wide range of applications. They offer a well defined behavior and the bottlenecks that can occur are also very well determined. In this sense the analysis of the application and the performance tuning on the fly can be carried out successfully. Using this environment the programmer can design its application in a quite simple way, and then he/she does not need to worry about any performance analysis or tuning, because the dynamic performance tuning takes care of these tasks automatically. The methodology has been tested on a real application and the preliminary results are very successful.

References

1. D. A. Reed, P. C. Roth, R. A. Aydt, K. A. Shields, L. F. Tavera, R. J. Noe, B. W. Schwartz: *"Scalable Performance Analysis: The Pablo Performance Analysis Environment"*. Proceeding of Scalable Parallel Libraries Conference, pp. 104-113, IEEE Computer Society, 1993.
2. W. Nagel, A. Arnold, M. Weber, H. Hoppe: *"VAMPIR: Visualization and Analysis of MPI Resources"*, Supercomputer, vol. 1 pp. 69-80, 1996.
3. A. Espinosa, T. Margalef, E. Luque: *"Integrating Automatic Techniques in a Performance Analysis Session"*. Lecture Notes in Computer Science, vol. 1900 (EuroPar 2000), pp. 173-177, Springer-Verlag, 2000.
4. Y. C. Yan, S. R. Sarukhai: *"Analyzing parallel program performance using normalized performance indices and trace transformation techniques",* Parallel Computing, vol. 22, pp. 1215-1237, 1996.
5. J. K. Hollingsworth, B. Buck: "*Paradyn Parallel Performance Tools, DynInstAPI Programmer's Guide",* Release 2.0, University of Maryland, Computer Science Department, April 2000.
6. J. Schaffer, D. Szafron, G. Lobe, and I. Parsons: "*The Interprise model for developing distributed applications*", IEEE Parallel and Distributed Technology, 1(3):85-96, 1993.
7. J. C. Browne, S. Hyder, J. Dongarra, K. Moore, and P. Newton. "*Visual Programming and Debugging for parallel computing"*, IEEE Parallel and Distributed Technology, 3(1):75-83, 1995.
8. J. Jorba, T. Margalef, E. Luque, J. Andre, D. X. Viegas: *"Application of Parallel Computing to the Simulation of Forest Fire Propagation"*, Proc. 3rd International Conference in Forest Fire Propagation, Vol. 1, pp. 891-900, Luso, Portugal, Nov. 1998.
9. J. C. S. Andre and D. X. Viegas: "A Strategy to Model the Average Fireline Movement of a light-to-medium Intensity Surface Forest Fire", Proc. of the 2nd Intemational Conference on Forest Fire Research, pp. 221-242, Coimbra, Portugal, 1994.