MATE: Dynamic Performance Tuning Environment*

Anna Morajko, Oleg Morajko, Tomàs Margalef, and Emilio Luque

Computer Science Department. Universitat Autònoma de Barcelona
08193 Bellaterra, Spain
ania@aomail.uab.es, olegm@aia.ptv.es,
{tomas.margalef,emilio.luque}@uab.es

Abstract. Performance is a key issue in the development of parallel/distributed applications. The main goal of these applications is to solve the considered problem as fast as possible utilizing a certain minimum of parallel system capacities. Therefore, developers must optimize these applications if they are to fulfill the promise of high performance computation. To improve performance, programmers search for bottlenecks by analyzing application behavior, finding problems and solving them by changing the source code. These tasks are especially difficult for non-expert programmers. Current approaches require developers to perform optimizations manually and to have a high degree of experience. Moreover, applications may be executed in dynamic environments. Therefore, it is necessary to provide tools that automatically carry out the optimization process by adapting application execution to changing conditions. This paper presents the dynamic tuning approach that addresses these issues. We also describe an environment called MATE (Monitoring, Analysis and Tuning Environment), which provides dynamic tuning of applications.

1 Introduction

Parallel/distributed systems offer high computing capabilities that are used in many scientific research fields. They facilitate the determination of the human genome, computing atomic interactions or simulating the evolution of the universe. So biologists, chemists, physicists and other researchers have become intensive users of applications with high performance computing characteristics. They submit applications to powerful systems to get their results as fast as possible. In this context, performance becomes a key issue. To satisfy user requirements, applications must reach high performance standards. An application is inefficient and useless when its performance is below an acceptable limit. Therefore, applications must not only be systematically tested from the functional point of view to guarantee correctness, but must also be optimized to ensure that there are no performance bottlenecks.

The optimization (or tuning) process requires a developer to go through the application performance analysis and the modification of critical application parameters. First, performance measurements must be taken to provide data about the application's behavior. This phase is known as monitoring and collects data related to the

* This work has been supported by the MCyT (Spain) under contract TIC2001-2592, by the European Commission under contract IST- 2000-28077 (APART 2) and has been partially supported by the *Generalitat de Catalunya* – GRC 2001SGR-00218.

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 98–107, 2004.

execution of the application. Then, the performance analysis of this information is carried out. It finds performance bottlenecks, deduces their causes and determines the actions to be taken to eliminate these bottlenecks. Finally, appropriate changes must be applied to the application code to overcome problems and improve performance. However, all these tasks are somewhat complicated, especially for non-expert users.

The classical approach to performance analysis is based on the visualization of program execution. Tools that support this approach show the execution of the application in graphical and numerical views [1, 2, 3]. Then, users must analyze generated views recognizing the most problematic performance regions, determining the causes of the bottlenecks and finally changing the application source code. To reduce developer effort and relieve them of such duties as analysis of graphical information and determination of performance problems, an automatic analysis has been proposed. Tools using this type of analysis are based on the knowledge of well-known performance problems. Such tools are able to identify critical bottlenecks and help in optimizing applications by providing suggestions to developers [4, 5, 6].

All mentioned tools involve developers changing a source code, re-compiling, relinking and restarting the program. They require a certain degree of knowledge and experience of parallel/distributed applications and hence are appropriate for developers rather than for such users as biologists, chemists, physicists or other scientists. To tackle these problems, it is necessary to provide tools that automatically perform program optimizations. A good, reliable and simple optimization tool performing automatic improvement could be profitable for non-expert users as well as for developers. Because of the complexity of the solution, there are not many tools that support automatic application optimization during run time [7, 8]. Moreover, they lean towards automated tuning, which requires certain changes to the application.

This paper addresses the problem of automatic and dynamic tuning of parallel/distributed applications. Section 2 presents this approach showing its fundamental concepts. Section 3 describes a dynamic tuning environment called MATE. Section 4 shows a catalog of tuning techniques that we investigated as part of our study. Finally, section 5 presents the conclusions of this study.

2 Dynamic Performance Tuning

The main goal of dynamic automatic tuning is to improve application performance by modifying its execution without recompiling or rerunning it. In this approach, the following steps can be distinguished: application monitoring, performance analysis and modifications of the running program. All of these must be performed automatically, dynamically, and continuously during application execution. The dynamic analysis and introduced modifications enables adaptation of the application behavior to changing conditions in the application itself or in the environment. Dynamic tuning appears as a promising technique that exempts non-experts or programmers from some of the performance-related duties. The most useful dynamic tuning is that which can be used to successfully optimize a broad range of different applications. It would be desirable to tune any application even though its source code and application-specific knowledge is not available. However due to incomplete information this kind of tuning is highly challenging and at the same time the most limited. The key question is: *what can be tuned in an "unknown" application*?

2.1 Tuning Layers

The answer to this key question can be found by investigating how an application is built. Each application consists of several layers: application-specific code, standard and custom libraries, operating system libraries, and hardware. An application is based on services provided by an operating system (OS). OS offers a set of libraries so that system users do not need to worry about low-level hardware details. The application uses the system calls to perform hardware/system-specific operations. Besides that, applications use standard libraries that support them with a variety of functions, e.g. higher level I/O, mathematical, string manipulation functions and so on. Additionally, applications may use custom libraries that provide domain-specific functionality, e.g. communication libraries, numerical methods, programming frameworks. These libraries insulate programmers from low level details as they offer a higher level of abstraction. Finally, each application contains application-specific implementation and consists of modules that solve a particular problem.

Considering OS and library layers, the tuning process is based on well-known features for them. By investigating particular OS and libraries it is possible to find their potential drawbacks and hence determine problems common to many applications. For each drawback, a tuning procedure can be identified. Optimizing the application code is the most complex and less reusable, due to the lack of application-specific knowledge. Each application implementation can be totally different and there may be no common parts, even though they may provide the same functionality. An application can be tuned if there is knowledge of its internal structure. Therefore, to optimize the application layer, dynamic tuning should be supported in some way with certain information about the application.

2.2 Approaches to Tuning

Considering the available knowledge, we have defined two main approaches to tuning: automatic and cooperative. In the automatic approach, an application is treated as a black-box, because no application-specific knowledge is provided by the programmer. This approach attempts to tune any application and does not require the developer to prepare it for tuning (the source code does not need to be adapted) and, therefore, it is suitable for tuning such layers as the operating system and libraries. We can find many general tuning procedures common to many applications. For each particular problem, all the necessary information, such as what should be measured, how it should analyzed, and what should be changed and when, can be provided automatically. The cooperative approach assumes that the application is tunable and adaptable. This means that developers must prepare the application for the possible changes. Moreover, developers must define an application-specific knowledge that describes what should be measured in the application, what model should be used to evaluate the performance, and finally what can be changed to obtain better performance. The cooperative approach is suitable for the application tuning layer.

2.3 Performance Analysis

Performance analysis examines application behavior based on the collected measurements, identifies performance bottlenecks, and provides specific solutions that over-

come these problems. Application behavior can be characterized by an analytical performance model. Such a model may help to determine a minimal execution time of the application or predict the application performance. Such a model can contain formulas and/or conditions that recognize a bottleneck and facilitate determination of the optimal behavior. As input, a model needs the measurements extracted from the application execution. Based on these and applying adequate formulas, the performance model can estimate the desired application behavior, e.g. the optimal value of some parameter. Finally, the application can be tuned by changing the value of that parameter.

To make the presented approaches to tuning (automatic and cooperative) homogeneous and to make optimization on the fly possible and effective, we concluded that the application knowledge should be described as the following terms:

- measure point – a location where the instrumentation must be inserted
- performance model – determines an optimal application execution time
- tuning point – the code element that may be changed
- tuning action – the action to be performed on a tuning point
- synchronization – policy determining when the tuning action can be invoked.

2.4 Dynamic Modifications of an Application

All phases of improving the application performance must be done "on the fly". To instrument the application without accessing the source code, the code insertion must be deferred till the application is launched. Modifications cannot require source code recompilation or restart. The technique that fulfills these requirements is called dynamic instrumentation. It permits insertions of a piece of code into a running program. Dynamic instrumentation was used in Paradyn [6] to build an automatic analysis tool. The Paradyn group developed a library called DynInst [9].

Considering DynInst's possibilities and our definition of application knowledge, we determined tuning actions that can be applied on the fly to a tuning point. A tuning point can be any point found by DynInst in the application executable (e.g. function entry, function exit). We consider the following to be tuning actions:

- function replacement – function calls are replaced with a call to another function
- function invocation – an additional function call is inserted at a specified point
- one-time function invocation – a specified function is invoked just once
- function call elimination – a specified function call is eliminated
- function parameter changes – the value of an input parameter is modified
- variable changes – the value of a particular variable is modified.

All modifications must be performed carefully to ensure that the application continues its execution correctly and does not crash. Therefore, each tuning action defines the synchronization policy that specifies when the action can be invoked in a safe manner. E.g. to avoid reentrancy problems, race hazards or other unexpected behavior, a breakpoint can be inserted into an application at a specific location. When the execution reaches the breakpoint, the actual tuning action is performed.

3 MATE

To provide dynamic automatic tuning of parallel/distributed applications we have developed a prototype environment called MATE (Monitoring, Analysis and Tuning Environment). For the purpose of our work we have made the assumption of targeting our tuning system to C/C++ parallel/distributed PVM [10] applications running on a UNIX platform. MATE performs dynamic tuning in three basic and continuous phases: monitoring, performance analysis and modifications. This environment dynamically and automatically instruments a running application to gather information about the application's behavior. The analysis phase receives events, searches for bottlenecks, detects their causes and gives solutions on how to overcome them. Finally, the application is dynamically tuned by applying a given solution. MATE consists of the following main components that cooperate among themselves, controlling and trying to improve the application execution:

- Application Controller (AC) – a daemon-like process that controls the application execution on a given host (management of tasks and machines). It also provides the management of task instrumentation and modification.
- Dynamic monitoring library (DMLib) – a shared library that is dynamically loaded by AC into application tasks to facilitate instrumentation and data collection. The library contains functions that are responsible for registration of events with all required attributes and for delivering them for analysis.
- Analyzer – a process that carries out the application performance analysis, it automatically detects existing performance problems "on the fly" and requests appropriate changes to improve the application performance.

Figure 1 presents the MATE architecture in a sample PVM scenario. In this example the PVM application consists of 3 tasks distributed on 2 different machines. When the Analyzer has been started, it distributes the AC to all machines where the application is running to control all the tasks. Once the AC is distributed the Analyzer receives from it information about the configuration of a virtual machine. The performance analysis is based on tunlets. Each tunlet is a shared library that implements the analysis logic for one particular performance problem. A tunlet uses the Analyzer's Dynamic Tuning API (DTAPI) to perform the performance monitoring and tuning of a program. Tunlets are passive components that drive the analysis by responding to a set of incoming events. The Analyzer provides a container that is responsible for managing a set of tunlets simultaneously. Tunlets provide the Analyzer with an initial set of measure points that are forwarded to all ACs. Next, the Analyzer asks the AC to start the application. The AC loads the shared monitoring library (DMLib) to the task memory that enables its instrumentation.

During execution, the ACs manage the instrumentation of each task. The shared monitoring libraries are responsible for delivering registered events directly to the Analyzer. When an event record is received, the Analyzer notifies the corresponding tunlet and this tunlet in turn finds bottlenecks and determines their solutions. By examining the set of incoming event records, the tunlet extracts measurements and then uses the built-in performance model to determine the actual and optimal performance. If the tunlet detects a performance bottleneck, it decides whether the actual performance can be improved in existing conditions. If this is the case, it then asks the Analyzer to apply the corresponding tuning actions. A request determines what should be

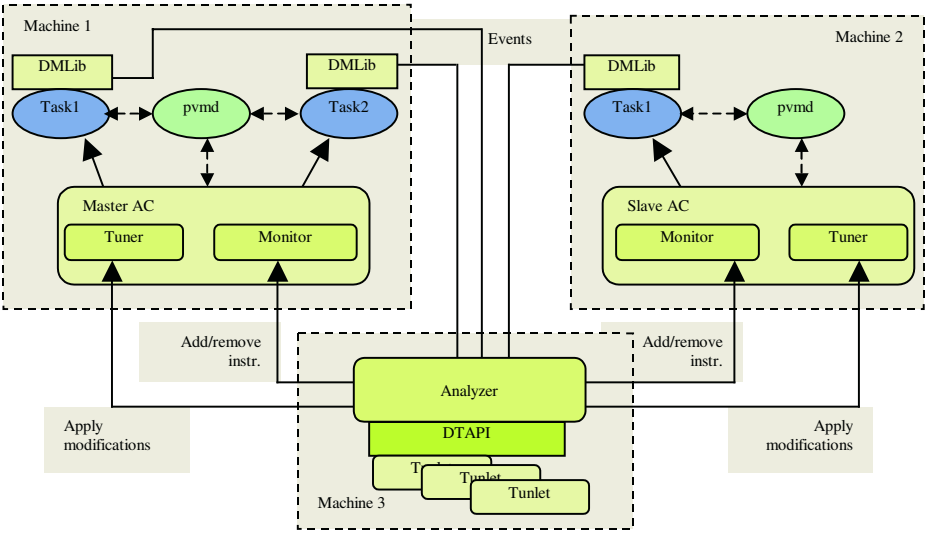

Fig. 1. Architecture of the MATE dynamic tuning for PVM.

changed (tuning point/action/synchronization) and it is sent to the appropriate instance of AC.

4 Tuning Techniques

We conducted a number of experiments on parallel/distributed applications to study how our approach works in practice. We proved that running applications under the control of MATE may be effective, profitable, and the adaptation of the application behavior to the existing conditions results in performance improvements. All required information related to one particular performance problem is what we call a tuning technique. Each tuning technique describes a complete optimization scenario:

- it specifies a potential performance problem of a parallel/distributed application
- it determines what should be measured to detect the problem (measure points)
- it determines how to detect the problem (performance model)
- it provides a solution for how to overcome the problem (tuning point/action/sync).

All experiments were conducted in a cluster of Sun UltraSPARC II workstations connected by LAN network. To investigate the profitability of dynamic tuning we used the following applications: (1) a set of synthetic master-worker programs; (2) Xfire – Forest Fire Propagation application [11], a computation-intensive program that simulates fireline propagation; (3) IS – NAS Integer Sort benchmark [12], a communication-intensive application that ranks integers using a bucket sort method.

We organized tuning techniques into a catalog in accordance with the tuning approach and the layer at which a modification occurs. Each tuning technique is implemented in MATE as a tunlet. Currently, we are focused on investigating tuning techniques separately. The catalog available in MATE is the following:

Operating system layer (Automatic approach):

- Message aggregation – minimizes communication overhead by grouping sets of consecutive messages into large ones. The tuning consists of replacing operating system function calls that transmit data, e.g. `write()`, with their optimized custom version with an aggregation mechanism. In networks with non-ignorable latencies and applications that send small messages, the technique can produce noticeable time savings (up to 91% in the case of synthetic applications).
- TCP/IP buffers – maximize the network transmission performance across high-performance networks using TCP/IP-based protocol. This is done by setting the send and receive socket buffers to an optimal value that can be estimated calculating bandwidth delay product. The tuning action includes one time system call invocation `setsockopt()` using `SO_SNDBUF` and `SO_RCVBUF` socket options. [13] reports improvements ranging from 17% up to 500% for FTP transmissions.

Standard library layer (Automatic approach):

- Memory allocation – improves performance by optimizing memory allocations. Programs that make intensive use of memory may benefit from optimized pool-based allocators if they perform a large number of small object allocations and deallocations. The tuning action replaces the standard allocator with the optimized pool allocator. The specialized pool allocators perform much better, giving up to 60-70% experimenting with synthetic applications.

Custom library layer (Automatic approach):

- PVM communication mode – minimizes the PVM communication overhead by switching the messaging to point-to-point mode. The tuning action includes one-time function call `pvm_setopt(PvmRoute,PvmRouteDirect)`. Changing the communication mode resulted in faster communication, up to 50% in synthetic applications and 17% in IS benchmark. The measured intrusion did not exceed 3,5% of the improved execution time.
- PVM encoding mode – minimizes the PVM encoding overhead by skipping data encoding/decoding phase. The tuning action includes input parameter modification of `pvm_initsend()` that changes the encoding mode from default XDR to data raw. We observed important benefits from data raw encoding mode (up to 74% in synthetic applications and up to 47% in IS benchmark). The intrusion reached up to 2,8% of the total application execution time.
- PVM message fragment size – selects the optimal size of message fragments to minimize the PVM communication time. The tuning action includes one-time function call `pvm_setopt(PvmFragSize,OptFragSize)`. To calculate the optimal fragment size, we used the experimentally deduced formula. This technique gave up to 55% profit in synthetic applications and up to 28% in IS benchmark. The intrusion reached 4,9%.

The PVM experiments that we conducted are described in more detail in [14].

Application layer (Cooperative approach):

- Workload balancing – balances the amount of work that is distributed by the master to workers considering the capacities and load of the machines where the application is running. During program execution, the simulation algorithm estimates the optimal workload. The tuning action changes the work factor by updating the variable value in the master that adapts the work assignment. This technique produced

up to 50% profit in synthetic applications and 48% in Xfire. The overhead was small – about 2% of the improved execution time.

- Number of workers – optimizes the number of workers assigned to perform a specified amount of work in the master/worker application. To calculate the optimal number of workers we used the performance model presented in [15]. The tuning action changes the number of workers by updating the variable value in the master process. We observed that changes to the number of workers could produce profits ranging from 20% to 300% in synthetic applications.

5 Related Work

The Autopilot [8] project bases on closed loop and allows parallel applications to be adapted in an automated way. It contains a library of runtime components needed to build an adaptive application. Autopilot provides a set of distributed performance sensors, decision procedures and policy actuators. The programmer can decide what sensors/actuators are necessary and then manually inserts them in the application source code. The toolkit includes fuzzy logic engine that accepts performance sensor inputs and selects resource management policies based on observed application behavior. Autopilot is similar to the MATE cooperative approach. However, it differs from the black-box approach where necessary measure and tuning points are decided and inserted dynamically and automatically. The Autopilot uses fuzzy logic to automate the decision-making process, while MATE is based on simple, conventional rules and performance models. Moreover, MATE is based on the dynamic instrumentation where measure and tuning points are inserted on the fly while in Autopilot it is done manually.

Active Harmony [7] is a framework that allows an application for dynamic adaptation to network and resource capacities. In particular, Active Harmony permits automatic adaptation of algorithms, data distribution, and load based on the observed performance. The application must contain a set of libraries with tunable parameters to be changed. Moreover, it must be Harmony-aware, that is, to use the API provided by the system. Active Harmony manages the values of the different tunable parameters and changes them for better performance. The project focuses on the selection of the most appropriate algorithm. This is conceptually similar to the cooperative approach of MATE. However, it differs from the automatic method that treats applications as black-boxes and does not require them to be prepared for tuning. Active Harmony automatically determines good values for tunable parameters by searching the parameter value space using heuristic algorithm. Better performance is represented by a smaller value of the performance function, and the goal of the system is to minimize the function. MATE uses a distinct approach where performance models provide conditions and formulas that describe the application behavior and allow the system to find the optimal values.

The AppLeS [16] project has developed an application-level scheduling approach. This project combines dynamic system performance information with application-specific models and user specified parameters to provide better schedules. A programmer is supplied information about the computing environment and is given a library to facilitate reactions to changes in available resources. Each application then selects the resources and determines an efficient schedule, trying to improve its own

performance without considering other applications. MATE is similar to AppLeS in that it tries to maximize the performance of a single application. However, it focuses on the efficiency of resource utilization rather than on resource scheduling.

6 Conclusions

Parallel/distributed programming offers high computing capabilities to users in many scientific research fields. The performance of applications written for such environments is one of the crucial issues. It is therefore necessary to provide good, reliable and simple tools that automatically carry out tasks involving performance analysis of parallel/distributed programs and behavior optimization. Our goal was to investigate and prove that dynamic tuning works, is applicable and may be effective. We also wanted to demonstrate that it is possible to support a user with a specific functioning environment for automatic dynamic tuning. Our work concluded with the prototype environment called MATE. It includes the monitoring, analysis and modifications of an application on the fly without stopping, recompiling or rerunning the application. The MATE environment tries to adapt the application to dynamic behavior. The conclusion of this work is that although dynamic tuning is a complicated task, it is not only possible, but also provides real improvements in application performance. This methodology seems to be a promising technique for accomplishing the successful performance of applications with dynamic behavior.

References

1. Heath, M.T., Etheridge, J.A. "Visualizing the performance of parallel programs". IEEE Computer, vol. 28, pp. 21-28. November, 1995.
2. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K. “Vampir: Visualization and Analysis of MPI Resources”. Supercomputer, vol. 12, pp. 69-80. 1996.
3. DeRose, L., Reed, D.A. "SvPablo: A Multi-Language Architecture-Independent Performance Analysis System". Proceedings of the ICPP 99, pp. 311-318. Japan, September, 1999.
4. Espinosa, A., Margalef, T., Luque, E. “Automatic Performance Analysis of PVM applications”. EuroPVM/MPI 2000, LNCS 1908, pp. 47-55. 2000.
5. Wolf, F., Mohr, B. ”Automatic Performance Analysis of Hybrid MPI/OpenMP Applications”. Euro PDP 2003, pp. 13-22. Italy, February, 2003.
6. Miller, B.P., Callaghan, M.D., Cargille, J.M. Hollingswoth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam K., Newhall, T. “The Paradyn Parallel Performance Measurement Tool”. IEEE Computer vol. 28. pp. 37-46. November, 1995.
7. Tapus, C., Chung, I-H., Hollingsworth, J.K. “Active Harmony: Towards Automated Performance Tuning”. SC’02. November, 2002.
8. Vetter, J.S., Reed, D.A. "Real-time Performance Monitoring, Adaptive Control, and Interactive Steering of Computational Grids". IJHPCA, Vol. 14, No. 4, pp. 357-366. 2000.
9. Buck, B., Hollingsworth, J.K. “An API for Runtime Code Patching”. University of Maryland, Computer Science Department, Journal of High Performance Computing Applications. 2000.
10. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V. “PVM: Parallel Virtual Machine, A User’s Guide and Tutorial for Network Parallel Computing”. MIT Press, Cambridge, MA, 1994.

11. Jorba, J., Margalef, T., Luque, E., Andre, J, Viegas, D.X. "Application of Parallel Computing to the Simulation of Forest Fire Propagation", Proc. 3rd International Conference in Forest Fire Propagation, Vol. 1, pp. 891-900. Portugal, November, 1998.
12. Bailey, D.H., Harris, T., Saphir, W., Wijngaart, R., Woo, A., Yarrow, M. “The NAS Parallel Benchmarks 2.0”, Report NAS-95-020, December, 1995.
13. “Automatic TCP Window Tuning and Applications”. Issue of NLANR Packets, http://www.nlanr.net/NLANRPackets/v2.1/autotcpwindowtuning.html August, 2000.
14. Morajko, A., Morajko, O., Jorba, J., Margalef, T., Luque, E. “Dynamic Performance Tuning of Distributed Programming Libraries”. LNCS, 2660, pp. 191-200. 2003.
15. César, E., Mesa, J.G., Sorribes, J., Luque, E. “Modeling Master-Worker Applications in POETRIES”. IEEE 9th International Workshop HIPS 2004, IPDPS, pp. 22-30. April, 2004.
16. Berman, F., Wolski, R., Casanova, H., Cirne, W, Dail, H., Faerman, M., Figueira, S., Hayes, J., Obertelli, G., Schopf, J., and Shao, G., Smallen, S., Spring, N., Su, A., Zagorodnov, D. “Adaptive Computing on the Grid Using AppLeS”. IEEE Transactions on Parallel and Distributed Systems, Vol. 14, No. 4, pp 369-382. April, 2003.