Dynamic Performance Tuning of Distributed Programming Libraries[1]

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

Computer Science Department. Universitat Autònoma de Barcelona.
08193 Bellaterra (Spain)
ania@aows10.uab.es, olegm@aia.ptv.es,
{josep.jorba, tomas.margalef, emilio.luque}@uab.es

Abstract. The use of distributed programming libraries is very common in the development of scientific and engineering applications. These libraries, from message passing libraries to numerical libraries, are designed in a very general way to be useful for a wide range of applications. Therefore, there are several polices that must be adapted to the particular application, system and input data to provide the expected performance. Our objective is develop an environment for tuning the use of a distributed library on the fly according to the dynamic behavior of the applications. In this paper, we present as an example a tuning environment for PVM-based applications. We show potential bottlenecks when using PVM. We also include tuning scenarios that describe the evaluation of the application behavior and the solutions that can improve the performance.

1 Introduction

High performance computing is provoking a new evolution in many fields of science and engineering. Physicists, biologists or chemists have become developers and end-users of high performance applications running on powerful systems (parallel systems, clusters, or even Grid systems). However, the design and development of such applications from scratch is a complex task that requires a high degree of expertise and deep knowledge of the system and the programming capabilities.

To insulate the programmer from the low level details, several libraries ranging from communication and message passing to numerical methods and programming frameworks have been developed. These libraries offer a higher level of abstraction and facilitate the design and development of high performance applications. However, these libraries are developed in a general way to be useful for a wide range of applications. Therefore, to accomplish the performance expectations, a developer must tune the library use by choosing the best polices considering application requirements and the environment. Such tuning process requires a deep knowledge of the library that is not necessary for developing applications. Moreover, these adaptations do not depend only on the application features, but also on the input data

[1] 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 partially supported by the Generalitat de Catalunya – GRC 2001SGR-00218.

or on the dynamically changing conditions of the application execution. Therefore, it is very hard to take into account all these variable conditions when developing applications. It is necessary then to tune the library usage on the fly during the application execution.

The goal is not to tune the library by modifying its source code, but to improve the way the library is used, e.g. switching a parameter in a library function call according to the variable environment conditions. This approach does not require analysis or tuning specific for an application, instead the optimization process is based on the features characteristic for the library.

The tuning is a complex and time-consuming task, and not feasible if it must be carried out manually by a developer. Therefore, it is beneficial to accomplish the performance expectations by using an automatic tuning environment. Such an environment is based on an expert knowledge of the possible bottlenecks and library limitations. It automatically inserts the required instrumentation to determine the current application behavior, decides what parameters must be modified and inserts the modifications dynamically to improve the performance of the whole application.

This approach is powerful because we focus on tuning the use of the library. Investigating the library it is possible to find its potential drawbacks. For each problem then the set of specific information can be determined, such as what should be measured to determine the behavior, which should be the desired behavior and which parameter should be modified to improve the application performance. The developers can concentrate on designing and developing an application, and submit it with the input data. After this the dynamic tuning environment takes care of controlling the application execution to ensure a good performance.

In the following sections of this paper, the dynamic tuning approach is presented. In Section 2, we introduce concepts and design of the MATE environment that supports dynamic tuning of parallel applications. Section 3 describes the aspects of PVM library considered as bottlenecks and presents their example optimizations. Section 4 shows practical experiments with dynamic tuning that we have conducted using MATE environment. Finally, Section 5 summarizes and concludes this work.

2 MATE

The main goal of dynamic tuning is to improve the application performance on the fly. This approach is suitable for applications with dynamic conditions, like variable behavior depending on the input data or variable behavior during the application execution. Running the application under control of a dynamic tuning system allows for adapting its behavior to the existing conditions. We propose a novel Monitoring, Analysis and Tuning Environment (called MATE) that provides dynamic automatic tuning of parallel applications. Our presentation of MATE focuses on its use with the PVM library [1].

MATE performs dynamic tuning in three basic and continuous phases: monitoring, performance analysis and modifications. This environment dynamically and automatically instruments and traces a running application to gather information about the application behavior. The analysis phase searches for bottlenecks, detects their causes and gives solutions on how to overcome them. Finally, the application is dynamically tuned by applying given solution. Moreover, the application does not need to be re-compiled, re-linked and restarted.

2.1 Basic Concepts

To achieve the objective set out by our work and allow for the application modifications on the fly, it is necessary to use dynamic instrumentation [2]. The basic idea is to defer program instrumentation until it is in execution and insert, modify and delete this instrumentation dynamically during run-time. MATE utilizes a special library called DynInst [3], [4] for two main purposes:

- Monitoring – to instrument an application during its execution. It is possible to add/remove a code that collects information about the application behavior.
- Tuning – to modify an application behavior. It is possible to change the code of the running application to improve its performance.

Since a parallel application consists of many intercommunicating processes physically executed on different machines, we distribute the modules of MATE to machines where the processes are running. To improve performance of the whole application, we gather information about all processes at a central location.

The tuning methods must be kept simple. The changes cannot affect the correct functioning of the application. We cannot assume that an application variable can be modified without taking any precautions. These factors limit the application of dynamic tuning, hence it is desired to provide external specifications of the programs behavior [5]. In MATE we assume the following terms: measure points, performance model, tuning points/actions. A measure point is a location in a process where the instrumentation must be inserted to provide measurements. A performance model consists of activating conditions (conditions in the application behavior considered to be a bottleneck) and/or formulas that allow for finding the optimal conditions. Tuning points are the elements that may be changed to improve application performance. Tuning action represents the action to be performed on a tuning point.

2.2 Design

As it is shown in Figure 1, MATE consists of three main components that cooperate among themselves controlling the execution of the application: a set of distributed Monitor modules (Monitors), a global Analyzer module (Analyzer), and a set of distributed Tuner modules (Tuners).

2.3 Monitor

The Monitors are distributed to all machines where a parallel application is running. They control the processes and collect the events produced during the execution. Each Monitor instance is responsible for a single machine. As PVM-based applications can create new processes and add new machines during execution, the Monitor integrates two special PVM services: the “tasker service” responsible for new process creation and the “hoster service” that controls machine management. The Monitor is able then to control virtual machine changes (add/remove host), as well as application processes (add/remove process). This information is always up-to-date and any changes are sent to the Analyzer module.

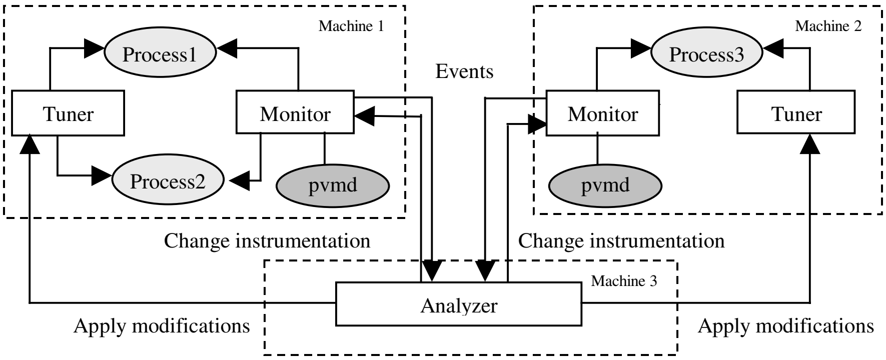

Fig. 1. Basic design of MATE in PVM environment.

To collect events, the Monitor dynamically inserts instrumentation into the original program execution at points needed to detect performance problems. Instrumentation generates events containing information about what happened, when and where. For example, if communication is monitored, the instrumentation can be inserted at the entry and/or exit of `pvm_send()` and `pvm_recv()` functions and each event should contain: timestamp, event type, source and target process identifiers, size of message. Generated events are sent to the Analyzer. This event flow can introduce high level of intrusion into the network and hence in some cases we aggregate the selected events. The Monitor is able to add or remove instrumentation dynamically. This capability minimizes the intrusion and permits to provide more precise information about the application behavior.

2.4 Analyzer

The Analyzer module is responsible for the automatic performance analysis of a parallel application “on the fly”. The Analyzer uses the knowledge (measure points, performance model and tuning points/actions) of possible problems and their solutions. All necessary information is specific for PVM problems and determined by the investigation of the library implementation. To be effective, the analysis is kept simple and decisions are taken in a short time.

When the Analyzer has been started, it receives from the Monitors information about the configuration of a virtual machine. This module is informed about all the changes in PVM virtual machine. The Analyzer chooses initial measure points and broadcasts them to all Monitors. When an application to be tuned starts, the Analyzer enters in a bottleneck search phase. It continuously receives requested events generated by different processes. By examining the set of events, the Analyzer extracts measurements. Then, it evaluates a performance model to determine the actual and optimal performance. When the Analyzer decides that the actual performance can be improved, it sends a request to the appropriate instance of Tuner, determining what should be changed, and where. For example, when the Analyzer determines that in a process a particular PVM function should be invoked with a specific parameter value, the name of the function, together with a new parameter value, is sent to the Tuner.

Obviously, during the analysis, Monitor modules are collecting and providing new data to the Analyzer. The Analyzer may need more information about program execution to determine the causes of a particular problem. It can therefore request the Monitor to change the instrumentation dynamically. The Analyzer also informs a user about detected problems and undertaken actions.

2.5 Tuner

The Tuner modules automatically change the application execution by inserting modifications into the running processes. The Tuners manipulate the process image in memory by means of DynInst library, hence they do not need to access a source code or restart the application. Similarly to the Tracers, the Tuners must have access to all application processes. Therefore, the Tuner modules are distributed among all the machines where the application is running. The Tuner module waits for requests from the Analyzer. The request specifies a target process, a tuning point and a tuning action. When the Analyzer detects a problem and its solution is found, a tuning request is sent to an appropriate Tuner instance. The Tuner receives it and applies given solution dynamically to a specified process or processes on the machine where it is running. This module contains a set of predefined modifications that can be activated by the Analyzer. We consider the following tuning actions, among others:

- One time function invocation – calling a specified function in the application.
- Function parameter changes – the value of an input parameter is modified before a function body is executed.
- Function invocation – an additional function call is inserted into the application at a specified point.
- Function replacement – all calls to a given function are replaced with a call to another function.

3 Tuning Examples

In this section, we present some of the PVM tuning examples that we studied within MATE. All examples illustrate aspects of the PVM library that can significantly improve the application performance. First, we investigate the library searching for its characteristics that can cause significant bottlenecks in an application performance. Then, for each potential bottleneck, we describe an optimization scenario, namely: what should be measured to detect the problem (set of measure points), how to discover the problem (performance model and activating conditions), and a solution as to overcome the problem (tuning point/action).

In the following subsections we present potential bottlenecks and their optimization scenarios when using PVM library: communication mode, data encoding mode, and message fragment size. To demonstrate communication performance of the

PVM environment and indicate its problems, we used a master-worker benchmark program that exchanges messages of various sizes. The presented measurements were repeated thousands of times, and the average round-trip time and its standard deviation (i.e. error) were calculated. Experiments were conducted in the environment consisting of a cluster of homogenous workstations Sun UltraSPARC II, 440 MHz connected by 100Mb/sec network.

3.1 PVM Communication Mode

PVM tasks have two modes to establish communication with other tasks, the task-to-task mode (direct) and task-to-daemon-to-daemon-to-task mode (indirect). By default, the indirect mode is used so all the messages are routed through the daemons. The direct mode is available only on some architectures. In this mode the tasks bypass the PVM daemon and have a direct link to each other using a separate socket. Although the initial TCP set up time is larger, all subsequent communication between the same two tasks is usually faster. The primary drawback of this method is that each TCP socket consumes one file descriptor, so there is a limit on maximum number of opened connections. This mode is less scalable, but it is a faster transfer method.

In Table 1 we compare the measurements of PVM communication performance in both communication modes. We can observe that the change from indirect to direct mode results in significantly faster communication (up to 50%). We must also point out that in majority of typical environments used to run PVM applications (workstation clusters) this mode is available, but rarely used.

Table 1. Benefits gathered from changing communication mode in a round trip application.

MsgSize [B]	Indirect Time [ms]	Direct Time [ms]	Difference [ms]	Average Benefit %
1	1,08 (±0,04)	0,53 (±0,02)	0,55 (±0,06)	50,72%
10	1,09 (±0,04)	0,54 (±0,01)	0,55 (±0,05)	50,39%
100	1,15 (±0,04)	0,61 (±0,02)	0,53 (±0,07)	46,47%
1000	1,77 (±0,05)	1,18 (±0,03)	0,59 (±0,08)	33,45%
10000	10,66 (±0,23)	8,51 (±0,21)	2,15 (±0,44)	20,13%
100000	104,95 (±4,11)	84,17 (±3,91)	20,78 (±8,02)	19,80%
1000000	1059,64 (±30,08)	861,37 (±39,07)	198,27 (±69,16)	18,71%

The application can configure the mode explicitly, but by default the indirect mode is used. During the execution, we can detect the use of indirect mode by calling `pvm_getopt(PvmRoute)` and check conditions to use direct mode. This mode is available when the environment does not include shared-memory machines and the number of PVM tasks is smaller than system-dependent limit. The tuning action includes one-time function invocation `pvm_setopt(PvmRoute, PvmRouteDirect)` that activates the mode. To avoid reentrancy problems in PVM library implementation, the invocation must be synchronized with the application execution. Therefore, first the breakpoint is inserted at the entry of function `pvm_send()` and when it activates the actual invocation is performed.

3.2 Data Encoding Mode

When PVM transfers the data, it converts the data format transparently between machines that have different architectures. By default PVM encodes data using XDR (external data representation) standard, because it cannot know if the user is going to add a heterogeneous machine before this message is sent. If there is no heterogeneous machine, the next message will only be sent to a machine that understands the native format. The encoding phase therefore, can be skipped, what allows for avoiding data encoding costs and reducing execution time.

In Table 2 we can observe the benefits from data raw encoding mode in comparison to default XDR encoding obtained with our benchmark application (up to ~74%). We can see that encoding overhead grows together with message size. For message sizes less than 1KB, the difference is low. If bigger amount of data is sent, more time is required to encode/decode it. We conclude that the data raw mode is significantly faster and preferable in the typical homogeneous clusters.

Table 2. Benefits gathered from changing data encoding mode in a round trip application.

MsgSize [B]	XDR Time [ms]	Data Raw Time [ms]	Difference [ms]	Average Benefit %
1	0,53 (±0,02)	0,51 (±0,01)	0,01 (±0,03)	2,25%
10	0,53 (±0,02)	0,52 (±0,01)	0,01 (±0,03)	2,25%
100	0,61 (±0,02)	0,58 (±0,01)	0,03 (±0,03)	4,76%
1000	1,17 (±0,02)	0,98 (±0,01)	0,19 (±0,04)	16,38%
10000	8,53 (±0,30)	3,28 (±0,05)	5,24 (±0,34)	61,49%
100000	84,69 (±4,02)	23,36 (±1,20)	61,33 (±5,22)	72,42%
1000000	873,90 (±40,35)	227,37 (±19,81)	646,52 (±60,16)	73,98%

The application can configure the data encoding mode explicitly, but by default the XDR mode is used. MATE controls the addition of hosts to the virtual machine (tasker/hoster services) so it is able to detect if an application is executed in the homogeneous cluster. During the execution we can decide to use data raw mode if the condition of hosts' homogeneity is fulfilled. The tuning action includes insertion of instrumentation (entry of `pvm_initsend()`) that changes the encoding mode from XDR to data raw (parameter mode set to `PvmDataRaw`). Moreover, when a new machine with different architecture is about to be added, the Analyzer module can request to restore the XDR mode.

3.3 PVM Message Fragment Size

In PVM messages exchanged by the tasks are composed without a maximum length. This is accomplished internally by dividing bigger messages into fixed-size blocks called fragments. PVM uses a default fragment size of 4KB (implementation limits it to 1024KB). When sending large messages, a number of fragments must be allocated and then separately sent. This causes high fragmentation what may reduce performance. Therefore, by changing message fragment size, bandwidth can be increased significantly. When the fragment size increases, PVM dynamically allocates more memory while sending/ receiving messages, hence more data is sent/received per system call. The drawback to this strategy is increased memory usage. It must be also pointed out that fragment size changes do not give significant effects in indirect communication mode, because they do not affect the behavior of PVM daemons.

The results of experiments are presented in Table 3. All experiments were conducted using direct communication mode. The default fragment size proved to be the optimal choice for small message sizes (less than 4KB). However, we can observe that for bigger messages the benefits from using larger fragment sizes are significant (up to 55%). This example demonstrates that the optimal fragment size depends on the application behavior – size of data that is sent and received. Moreover, the optimal value can vary during the execution (due to program phases) and hence it is not enough to calculate it once, but it should rather be adapted to the application behavior.

Table 3. Benefits gathered by changing the message fragment size in a round trip application.

FragSize	4 KB	16 KB		64 KB		256 KB		512 KB	
MsgSize	Time [ms]	Time [ms]	Benefits [%]	Time [ms]	Benefits [%]	Time [ms]	Benefits [%]	Time [ms]	Benefits [%]
2 KB	1,77	1,75	1,11	1,77	-0,22	1,76	0,34	1,79	-1,45
4 KB	4,54	2,51	44,68	2,51	44,68	2,51	44,80	2,50	44,88
8 KB	7,99	4,13	48,37	4,14	48,16	4,25	46,85	4,12	48,45
64 KB	55,64	34,11	38,70	29,47	47,03	27,08	51,33	26,85	51,74
256 KB	226,45	133,08	41,23	113,31	49,96	107,99	52,31	104,93	53,66
512 KB	460,66	264,40	42,60	224,41	51,28	214,75	53,38	212,99	53,76
1024 KB	920,62	525,61	42,91	443,56	51,82	426,94	53,62	425,02	53,83
4096 KB	3730,53	2105,77	43,55	1739,84	53,36	1666,26	55,33	1657,24	55,58

The application can configure the fragment size explicitly, otherwise the default value is used. During the execution, we can query the actual size by calling `pvm_getopt(PvmFragSize)` and detect if the application changes this value by instrumenting the function `pvm_setopt(PvmFragSize, size)`. Currently, to calculate the optimal fragment size, we use the experimentally deduced formula:

OptimalFragSize = Average (message size) + Std deviation (message size)

The Analyzer module requires the number and sizes of transmitted messages. Therefore, the communication calls (e.g., `pvm_send()`, `pvm_recv()`, `pvm_mcast()`) must be instrumented to gather statistics. The analysis of collected data is performed periodically and separately for each task. However, the tuning action is not applied each time the new optimal value is calculated. Instead, the tuning action is triggered when the difference between current and optimal values exceeds a fixed threshold, and the estimated communication cost becomes significant during the period. The tuning action includes one-time function invocation `pvm_setopt(PvmFragSize, OptimalFragSize)` that changes the current fragment size. The invocation must also be synchronized as described in Section 3.1.

4 Tuning Experiments

This section presents the results of practical experiments with dynamic tuning that we conducted using MATE. We provide timings that characterize the performance obtained executing parallel application with our environment MATE. We evaluate dynamic tuning costs, i.e. the overhead incurred by MATE in comparison to traditional PVM implementation. The total intrusion time includes application startup (process load time, process image parsing time), instrumentation (adding, removing, execution and reporting), analysis and tuning. Experiments show performance improvements originated from dynamic tuning when applying different scenarios. Each test was performed hundreds of times and the average of the wall clock execution times of the master process was taken. We show that tuning is effective and benefits are higher than the overhead introduced into the application execution.

4.1 NAS Parallel Benchmark – IS Kernel (Sort Application)

To conduct our experiments, we selected a computationally-intensive parallel program. We used Integer Sort (IS) kernel benchmark from NAS Parallel Benchmark suite [6]. The IS kernel ranks a large array of small integers as fast as possible using a bucket sort method. Communication costs are high (up to about 50% communication) in this application and are dominated by all to all data exchange messages, wherein each processor sends to all others this data which falls in the range of the recipient.

Table 4. Execution time of IS Kernel Benchmark in different tuning scenarios.

No.	Tuning scenario	Execution time [sec]	Tuning benefits [sec]	Intrusion [sec]
1.	PVM (no tuning)	732	-	-
2.	PVM + communication mode tuning	604	127 (17,5%)	21 (~3,5%)
3.	PVM + data encoding mode tuning	761	-29 (-3,9%)	21 (~2,8%)
4.	PVM + message fragment size tuning	769	-37 (-5,1%)	27 (~3,5%)
5	PVM + all scenarios	529	202 (27,7%)	28 (~5,3%)

Table 4 presents the results of the IS Kernel Benchmark experiments in different tuning scenarios. In the first scenario the application was executed under standard PVM 3.4 without any tuning and it was used as a reference result. The other tests were performed under PVM, but MATE monitored and tuned the application.

In the second scenario, the PVM communication mode was tuned by MATE. The MATE analyzer decided to use the direct mode, as by default the application used indirect mode and our experiments were conducted in a small NOW environment. We can observe 17,5% benefit in execution time caused by this tuning action. Such improvement can be explained by high computation-communication ratio (1:1). The measured intrusion did not exceed 3,5% of the total execution time.

In the third scenario, tuning the data encoding mode was tried. The analyzer did not perform any tuning actions. Originally the application used data raw encoding mode, so no improvement was possible. However, when the application was experimentally executed in XDR mode, the improvement reached 47% (as the exchanged data was integers). The intrusion resulted in 2,8% of the execution time.

In the forth scenario, MATE conducted message fragment size tuning. In this scenario the analyzer requested the instrumentation of send and receive primitives in all of the tasks to examine number and sizes of messages. The analysis indicated that the default fragment size was improper, because each of the tasks was sending a series of very small messages (4B and 16B) as well as big messages (over 1MB). The requested tuning action increased the fragment size. After the change, the application remained stable and no more tuning was performed. However, we cannot observe any benefits, because by default indirect mode was used. The benefits are significant when conducting the same experiment in direct mode. The intrusion reached 3,5%.

Finally, in the fifth scenario, we conducted all described tuning scenarios in the same execution. Both communication mode tuning and fragment size tuning was applied successfully. Despite of the intrusion (about 5,3% resulted from more inserted instrumentation and higher volume of collected measurements), the introduced changes produced the best results, improving the total execution time up to 27,7%.

5 Conclusions and Future Work

Dynamic automatic performance tuning of distributed libraries appears as a powerful technique to accomplish a performance improvements of applications with a dynamic behavior. A complete dynamic automatic performance tuning environment (MATE) has been presented. MATE includes the monitoring, analysis and modifications of the application on the fly without stopping, recompiling or rerunning the application. In this paper we have focused on the tuning of PVM-based application. We have shown a set of problems that can appear as bottlenecks when using PVM library. We have proven that MATE is promising and is able to improve the performance of the application implemented in PVM by tuning the library usage. Obviously, environment causes intrusion and it is inserted into the application execution, but it is still smaller than the benefits obtained from the performed improvements.

The presented methodology is general and can be applied to improve the use of other libraries. Moreover, the set of conducted experiments gave us new ideas on future work and tuning examples. Many applications highly use memory. Therefore, it would be profitable to optimize memory usage by providing custom allocators as general-purpose allocators may be inefficient or inflexible. The tuning of communication protocol options correlated with message exchange can be beneficial as well. We also found interesting the tuning of work size unit or data distribution as it can reduce significantly the execution time.

References

1. 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.
2. Buck, B., Hollingsworth, J.K. *"An API for Runtime Code Patching"*. University of Maryland, Computer Science Department, Journal of High Performance Computing Applications. 2000.
3. Hollingsworth, J.K., Buck, B. *"DyninstAPI Programmer's Guide. Release 3.0"*. University of Maryland, January 2002.
4. Paradyn Project *"Paradyn Parallel Performance Tools, User's Guide, Release 3.3"*. University of Wisconsin, Computer Science Department, January 2002.
5. César, E., Morajko, A., Margalef, T., Sorribes, J., Luque, E. "*Dynamic Performance Tuning Environment Supported by Program Specification*". Scientific Programming, 10, pp. 35–44. 2002.
6. 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.