Universitat Autònoma de Barcelona

Dynamic Tuning of Parallel/Distributed Applications

Departament d'Informàtica
Unitat d'Arquitectura d'Ordinadors
i Sistemes Operatius

Thesis submitted by Anna Morajko in fulfillment of the requirements for the degree of Doctor per la Universitat Autònoma de Barcelona.

Barcelona, December 18, 2003

Dynamic Tuning of Parallel/Distributed Applications

Thesis submitted by Anna Morajko in fulfillment of the requirements for the degree of Doctor per la Universitat Autònoma de Barcelona. This work has been developed in the Computer Science department of the Universitat Autònoma de Barcelona and was advised by Dr. Tomàs Margalef Burrull.

Thesis advisor

Tomàs Margalef Burrull

Barcelona, December 18, 2003

Acknowledgements

First of all I want to thank to my advisor, Dr. Tomàs Margalef, for giving me the chance to make such a work. I thank him for spending with me so much time and helping me with my research. I appreciate his contribution and suggestions resulting in significant improvements of my work. Moreover, I really thank him for his encouragement in my private life during all these years.

I thank to Prof. Emilio Luque for his continuous support, contribution and valuable criticism and suggestions. Thanks to him I had so many chances to present my work to the public. His life understanding made that this work has been completed.

I would like to express my thanks to Barton Miller for his ideas and continuous questions concerning the work. The expert support and so quickly given answers were really appreciable.

I also want to thank a lot to all people from the CAOS group. They facilitated me every day of my work. In particular I want to thank to Anna Cortés, Elisa Heymann, Eduardo César, Remo Suppi, Josep Jorba for their emotional support and disposition at any time.

I thank very much to my family. First of all, my deepest thanks go to my husband, Oleg Morajko, for the most important thing in my life: for being with me all the time for better and worse. I thank him for his help, for uncountable advises and suggestions, and for his really significant technical support. My great thanks go to my parents, I really appreciate their understanding, constant encouragement and big help at home. Finally, I would like to give my thanks to my sweet son Dawid for being so good boy what helped me to finish the work.

Table of contents

Abstract

The main goal of parallel/distributed applications is to solve the considered problem as fast as possible utilizing a certain minimum of the parallel system capacities. In this context, the application performance is one of the most important issues. The classical way of improving the application performance is based on the analysis of the monitoring information obtained from an execution of the application. Developers must search through this information for the bottlenecks and optimize the application behavior changing the source code manually. This approach requires developers to do many tasks and have a great experience about parallel programming. Therefore, the classical application tuning is then very difficult especially for non-expert programmers. It is necessary to provide tools that automatically carry out these tasks. Moreover, this classical approach is not feasible when the applications have a dynamic behavior. Many applications have a different behavior according to the input data set or even change their behavior dynamically during the execution. In this case, another approach is required to accomplish performance expectations. It would be desirable that the performance tuning could be done on the fly by modifying the application according to the particular conditions of the execution.

This thesis addresses the problem of automatic and dynamic tuning of parallel and distributed applications. Our objective is to help developers in the process of improving the application performance. This work presents a whole solution that deals with the issues of automatic and dynamic application improvement. In this approach, an application is monitored, its performance bottlenecks are detected, solutions are given and the application is modified on the fly. All these steps are performed automatically, dynamically and continuously during application execution. This approach exempts developers from performance analysis and difficult intervention to a source code by automatically improving the performance of parallel programs during run-time. The dynamic analysis and introduced modifications permits to adapt the behavior of the application to dynamic variations.

With this objective we have developed an environment called MATE (Monitoring, Analysis and Tuning Environment) that provides dynamic automatic tuning of parallel applications. 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, while it is being tuned, the application does not need to be re-compiled, re-linked and restarted.

Many various practical experiments have been conducted on distributed and parallel applications to see if this approach really works. We have proven that it is effective, feasible, profitable, and can be used for a real improvement of the program performance. Running applications under control of a dynamic tuning system has allowed for adapting their behavior to the existing conditions and improving their functionality.

Chapter 1

Introduction

In this chapter, we present a general overview of the application performance problem. In particular our work focuses on parallel and distributed applications. We show what are the goals of this thesis and its contributions. Finally, we present the organization of thesis.

1.1. General overview

The high performance computation demand increases day by day. In many different fields, but in particular scientific, have appeared a strong need of such a computation since each time more and more problems must be solved by specially developed applications. This situation has taken the evolution of science to a new step called computational science. Applications that support computational science facilitate the determining of the human genome, computing the atomic interactions in a molecule, simulating the evolution of the universe or climate model simulation, to mention only few. So biologists, chemists, physicists and many other researchers have become intensive users of applications with high performance computing requirements. The usage of such applications requires many resources as they become more data intensive and perform more sophisticated calculations. Therefore, scientists submit very large applications to powerful systems in order to solve the problems and get the results as fast as possible, considering the largest data size and taking advantage of all the resources available in the system.

The increasing need of high performance systems has driven scientists towards parallel/distributed systems. Parallel systems are computers consisted of a set of processing units that work cooperatively in parallel to solve a computational problem. Parallelism is a general term used to characterize a variety of simultaneous actions occurring in a computer, especially on a parallel computer. Parallel computers offer high potential resources like processing speed, memory or disk capacity. Although such systems have their performance limits, they are much more powerful than the rest of the computers and hence are better for solving scientific problems demanding intensive computation.

Therefore, architectures, compilers and operating systems have been developed to extract and use as much parallelism as possible in order to speedup computation.

Parallel applications must be developed in a specific way to be able to run on parallel systems and utilize their features. First of all such applications must provide the ability to perform many different operations at the same time. The main goal of these applications is to solve the considered problem as fast as possible utilizing a certain minimum of the parallel system resources. In this context, parallel application performance becomes a crucial issue. The difference between the expected and real performance should not appear as a significant gap. The objective is to reduce this gap as much as possible.

Therefore, programmers of parallel applications are responsible for providing the best possible behavior of these applications. Applications will be useless and inappropriate when their performance is under acceptable limit. Programmers then face up to many problems that must be solved if such applications are to fulfill their promises to obtain the highest performance in a due environment.

Once an application has been implemented in parallel, developers must systematically test it from the functional point of view to guarantee its correctness. Then, to reach the goal and provide the highest performance, programmers are obligated to carry out an application optimization process to ensure that there are no performance bottlenecks during the application execution.

The optimization process, also known as tuning process, requires a developer to go through the application performance analysis and the modification of critical application parameters. The tuning process implies then several phases. First, the performance measurements must be taken in order to provide information about the application. This phase is known as the monitoring – it collects information related to the execution of the application. Then, the analysis of this information is performed. Performance analysis finds performance bottlenecks, deducts their causes and determines the actions to be taken to eliminate these bottlenecks. Finally, appropriate changes must be applied into the application code to overcome problems and improve performance.

The essential, and at the same time the most complicated task of the tuning process is performance analysis. It must be pointed out that, in practice, the causes of performance bottlenecks can be found at different levels. For example, a communication problem can result from:

- An erroneous conception of the application that provokes an unnecessary blocking time in a receive primitive.
- Communication library implementation. In many cases, the design or implementation of the software layers is generic and is not optimized for a particular system or for particular conditions. This implies that the application may behave differently than expected.
- Operating system features. For example, an inappropriate buffer size and the message treatment at the protocol level can interfere with application message delivery times.
- Underlying hardware capabilities. The interconnection network features (latency, bandwidth, etc.) or even the contention in the network can seriously slow down the application.

As a consequence, the developers are forced to master the application, the involved software layers and the distributed system behavior. Moreover, parallel computing evolves from homogeneous parallel systems to distributed heterogeneous systems what significantly extends difficulties. In many cases the application performance also depends on the input data set. This fact implies that a set of potential bottlenecks can vary for different executions. All these issues make the performance tuning process difficult and costly, especially for non-expert programmers as it requires a high degree of expertise to really improve the performance of the application.

To tackle all these problems, user-friendly tools should be available. However, in the area of performance optimization, there is still a lack of real useful tools and most of them require from the user a deep knowledge about the parallel and distributed systems and programming. Therefore, it is necessary to provide tools that automatically carry out tasks of the parallel program optimizations what exempts a developer from some of the performance-related duties. The required tools include programming environments, debugging tools and performance tuning systems. A good, reliable and simple performance optimization tool is necessary to provide a developer with appropriate and sufficient information about the application behavior, as well as with possible changes. Such a tool could help a programmer to improve the performance of the application.

1.2. Our goals

The principal objective of this work is to investigate if it is possible to dynamically tune performance of distributed parallel programs. In this sense, our idea is to automate all the phases that we have distinguished in the performance tuning process and perform them dynamically on the running parallel application. Such approach would help programmers in the whole process of improving the performance of parallel and distributed applications.

To support developers with dynamic performance tuning and practically evaluate its profitability, we would like to create the dynamic tuning environment that facilitates monitoring, performance analysis and optimization of parallel applications. All these steps should be done automatically, dynamically, and continuously during application execution. Therefore, our main idea is to build an environment that is characterized as shown in Figure 1.1. The ideal solution would be to construct a tool that is able to automatically accelerate the application execution by adapting it to changing conditions. Such a solution would relieve developers from the complex manual tuning tasks. Moreover, while performing these optimization phases it would be desirable not to require the access to the source code, recompilation, nor re-linking and restart. To ensure the profitability of the runtime optimizations, we must consider intrusion related issues and try to minimize its impact. Therefore, overhead introduced by the tuning tool must be small and not significantly interfere the application execution.

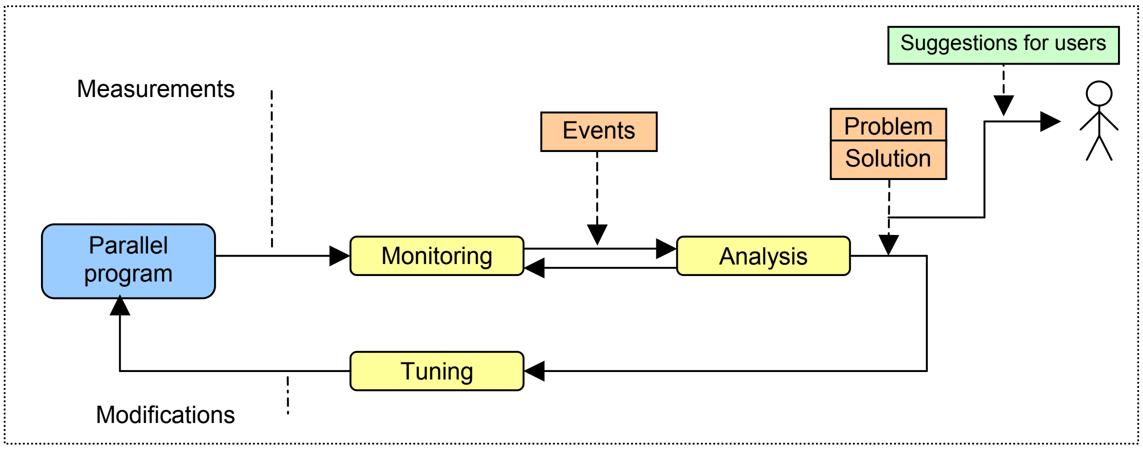


Fig. 1.1. Dynamic tuning approach provided by our environment.

Another goal of this thesis is practical experimentation with dynamic tuning in order to check its effectiveness and profitability. Therefore, we should conduct many tests on different real applications in real environment. Only experiments with existing tuning tool that permits dynamic optimizations of real applications will provide the complete view of the potential benefits obtained when using this approach.

An important issue that should be also investigated is the applicability of the dynamic tuning approach. We have to verify in what circumstances dynamic tuning can be applied effectively and what conditions must be fulfilled by parallel applications and systems. The question to be answered is if dynamic tuning can totally replace other – classical and automatic – approaches to the performance analysis. Is it a solution for all the problems that appear in the optimization process? Or is it in some cases limited? If it is the case we must define the limits of its usage.

An issue of particular importance is the representation of knowledge that we can utilize when optimizing an application. A tuning tool must have built-in such a knowledge to use it while analyzing the application behavior, finding bottlenecks and determining solutions. Application analysis without knowledge about its internal structures and dynamic modifications of unknown application structures is very complicated. It is not realistic to assume that any modification on any application in any environment can be done on the fly. This knowledge should be specified independently from the tool implementation, in order to permit extensibility and the inclusion of new performance problems. It would be ideal to provide totally external solution that would support the tuning tool with all the required information about the application.

1.3. Contribution

The main objective of this thesis has been to show that the performance of distributed parallel programs can be improved automatically during run-time. Therefore, we have investigated this idea and it has given a fruitful results that can be summarized in that this approach works, is applicable, effective and useful in certain conditions. We have proved that it appears as a powerful technique to accomplish the successful performance of applications with dynamic behavior.

However, we have encountered some limits of the usage of this approach. Due to incomplete application information dynamic tuning of unknown application is complicated, hard or sometimes even impossible. In general the performance analysis cannot be performed effectively without knowledge about what the application does. Dynamic modifications of unknown application structures are complex, may appear as dangerous and hence must be done very carefully. Therefore we have distinguished different layers of the application that can be tuned: application-specific code, standard and custom libraries (API + code), operating system libraries (API + code), hardware. For some layers we have many common information and hence we can extract well defined bottlenecks representative for many applications and define their solutions. In other case, it is required to provide a knowledge about the specific application problems and solutions since there is no information about the potential application bottlenecks. We differentiated two tuning approaches: automatic and cooperative. In the automatic approach the application is treated as a black-box, because no application-specific knowledge is provided by the programmer. In the cooperative approach we assume that an application is tunable and adaptable as a developer must provide application-specific information and prepare an application for the possible changes.

To make the solution homogeneous for both the automatic and cooperative tuning approach, we decided that the application should be represented by a set of necessary information required for the monitoring, analysis and tuning. We defined that the application knowledge consists of measure points (what must be monitored in the application), performance model (activating conditions and/or formulas that allow for finding the optimal conditions), tuning action/point/synchronization (what, where and when can be changed in the application to obtain its better performance).

Another principal contribution of the thesis has been a real functioning tool that supports users with automatic and dynamic tuning and relieves them of many complex tasks. Moreover, we also wanted to prove experimentally that dynamic tuning is useful, beneficial and applicable. For this purpose we have developed MATE – Monitoring, Analysis and Tuning Environment. MATE supports three basic functionalities: performance monitoring, performance analysis and tuning. All these phases are performed automatically and continuously on running parallel distributed applications. The environment is based on the computational steering loop concept and exempts a developer from intervention into the tuning process. Conducted experiments showed that MATE is able to adapt the application to the dynamic behavior and can be applied to many performance bottlenecks that may appear during the application execution.

To perform all phases on the fly, we based our environment on the dynamic instrumentation. This technique provides a possibility to manipulate a running program without access to its source code. By applying this method, it is possible to monitor and tune a parallel program during run-time. Moreover, the program does not need to be re-compiled, re-linked and restarted while instrumenting and applying changes. MATE is based on the DynInst library to provide the performance monitoring (application instrumentation and data collection) and tuning (modifying the code of running application).

MATE provides both black-box and cooperative tuning. There are some common bottlenecks that MATE is able to monitor, detect and solve automatically, but it also provides an easy way to add information about other performance problems. The environment is based on the knowledge that contains a set of tuning techniques representing different problems. To support the analysis of many problems, MATE includes the catalog of tuning techniques where each technique solves a particular problem. One tuning technique provides information about measure points, performance model (analytical model or set of rules) and tuning action/points/synchronization. Such a knowledge is provided to MATE via specific libraries called tunlets.

1.4. Organization of thesis

The work presented in this thesis is divided in chapters as follows.

Chapter 2 introduces the general overview of the classical approach to the performance analysis of parallel applications. It describes the measurement techniques used for the purposes of this approach. Finally we present a set of tools that support the classical performance analysis.

In Chapter 3 we describe the principles of the second approach to the performance analysis, namely automatic analysis. We also introduce an extension to this approach which permits dynamic analysis of the parallel application during their execution. Finally, we show the set of tools that provide automatic performance analysis.

Chapter 4 is devoted to the dynamic automatic performance tuning approach. We explain the fundamental concepts of dynamic optimization and introduce its requirements and constraints. This chapter compares dynamic tuning to automatic and dynamic analysis and indicates advantages and disadvantages of both approaches. We define our principal taxonomy and classification of dynamic tuning and we show at which layers the performance tuning is possible. We present our representation of knowledge that is required to dynamically optimize an application. This chapter also shows our idea on how to facilitate dynamic tuning process. Therefore, we present the design of pattern-based framework that provides parallel application specification. We introduce the specific library called DynInst, that supports dynamic instrumentation approach what allows us for application monitoring and optimization on the fly. We introduce the set of modifications that are possible to be applied dynamically. Finally, this chapter presents a set of existing tools that are related with the tuning area.

In Chapter 5 we present design and implementation of MATE – Monitoring, Analysis and Tuning Environment. We describe our dynamic tuning environment presenting its requirements, architecture and design. We show all its modules namely Monitor, Analyzer and Tuner together with the implementation aspects.

Chapter 6 presents a catalog of tuning techniques that we studied within our work. Each tuning technique is described in a systematic way and consists of a set of sections that explain the performance problem the technique addresses, its general applicability, solution it applies, the implementation aspects and the conducted experiments. The experimental work provides the possibility to see the benefits that we can achieve utilizing MATE to improve the application performance. We collected all measurements by tuning synthetic, as well as real applications.

Finally, Chapter 7 summarizes and concludes our work, outline open problems and discuss directions for future work.

Chapter 2

Classical performance analysis

In this chapter, we present a general overview of the classical performance analysis of parallel and distributed applications. Two principal approaches to the classical application analysis are described: based on visualization of the execution and based on prediction of the behavior. We introduce different existing techniques of performance measurement that are utilized in the performance data collection process for the purposes of the application behavior analysis. We also present a set of available tools that support classical performance analysis of parallel programs.

2.1. Approaches

The principal goal of parallel and distributed applications is to benefit from the potential high computational capabilities of parallel systems. However, obtaining high performance of an application running in such a system becomes a hard task. To develop an application characterized by adequate performance, programmers must face up the analysis process. Therefore, to attend the performance analysis problem and help programmers in the application improvement, many tools have been presented. There are two principal classical solutions implemented by these tool: the first has been classified as "measure and modify", the second as predictive.

The "measure and modify" approach of the classical analysis of parallel applications is the oldest one and is based on the visualization of the program execution. Generally, tools that support this approach show the execution of the application in different graphical and numerical views. To be able to visualize application behavior, first the classical tools requires the usage of monitoring tools to obtain performance data (a.k.a. measurements) from the application execution. Then, visualization tools perform measurements and generate different graphics of application behavior. As the next step, users must analyze generated views selecting the most problematic regions and finally change the application source code. This process repeats again until an adequate performance is achieved. In this approach two main steps can be identified: monitoring and visualization.

On the other hand, the principle of the predictive approach to the classical performance analysis is to build a model of the application behavior. Such a model provides a way to understand performance problems. The predictive analysis is based on the simulations of the application execution. As a result of the performance analysis a user will receive the expressions constructed by the tool that model the application performance. Using these models benefits in the prediction of the future application behavior. However, the user must understand and process the performance analysis results to improve the application.

2.2. Performance monitoring

The application performance analysis requires performance data gathered from the application executions. Therefore, the application must be monitored in order to get such a data. The performance monitoring process consists of two main phases: instrumentation and measurement collection. The parallel program is executed under control of a monitoring tool that allows for measuring and collection of performance data. The main purpose of this data is to illustrate specific information about the application execution. To generate such a data, some piece of logging code, so called instrumentation, must be inserted into the original code of the application. The instrumentation is inserted at all points in the application code that are to be monitored. In the classical analysis approach the insertion is done by the monitoring tool in preprocessing phase or manually by the programmer. Once the application has been instrumented with the specific calls to the monitoring code, it must be compiled and linked with appropriate monitoring libraries.

It is also possible to insert the instrumentation dynamically. The principle of dynamic instrumentation is to defer program instrumentation until it is in execution and insert, alter and delete this instrumentation dynamically during program execution. The program being modified is able to continue its execution and does not need to be re-compiled, re-linked, or restarted. Dynamic instrumentation is provided by a library called DynInst. Details of dynamic instrumentation and the DynInst library will be explained later in Chapter 4.

When the application is being executed and is performing a part of code with inserted instrumentation, this instrumentation allows for data measurement and collection. For example, to calculate how many times a function is called, the monitoring call must be inserted at the beginning of this function. Then, during the application execution, the monitoring code, when invoked, will increment an internal counter. Phases of instrumentation and measurement collection in the classical performance analysis are shown in Figure 2.1.

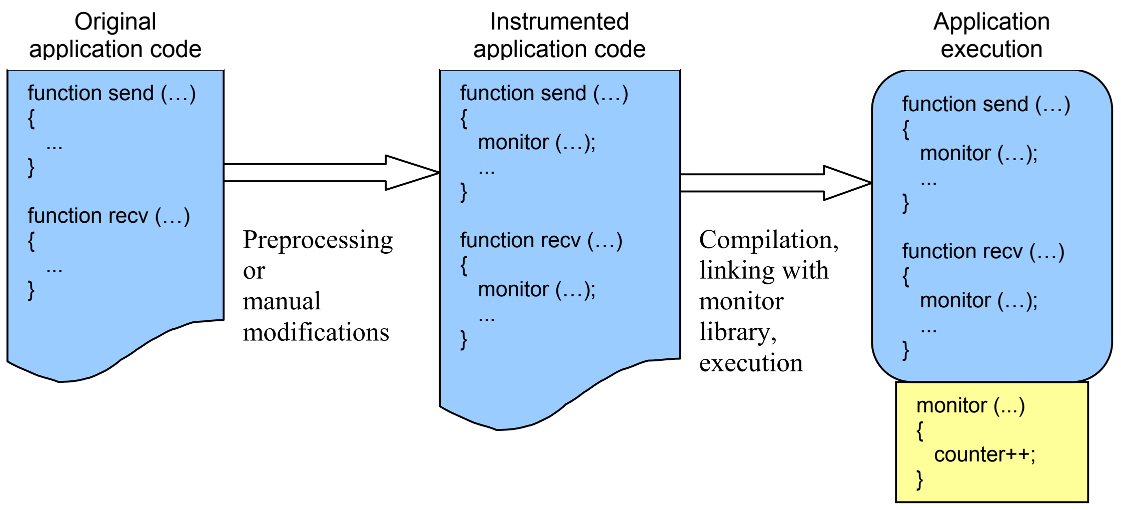

Original application code
function send (…) { … }
function recv (…) { … }
Preprocessing or manual modifications
Instrumented application code
function send (…) { monitor (…); … }
function recv (…) { monitor (…); … }
Compilation, linking with monitor library, execution
Application execution
function send (…) { monitor (…); … }
function recv (…) { monitor (…); … }
monitor (...) { counter++; }

Fig. 2.1. Classical monitoring process that consists of instrumentation and measurement collection.

As indicated in [Ree93] there are three basic approaches to performance data capture: timing, profiling (counting and sampling) and tracing. Each measurement technique represents a different balance between the amount of information, potential perturbation, accuracy and implementation complexity. In the following subsections we present a short description of different monitoring techniques.

2.2.1. Timing

This technique relies on a measure of execution time. The measurements are performed by specific calls to timing libraries. Such a library may be based on the following timing function calls: `clock()`, `times()`, `gettimeofday()`, `gethrtime()`, `getrusage()`, `MPI_Wtime()`, `Fortran90 qw_time()`, `system_clock()`. Calls to the timing library must be inserted into the application code. These calls collect in execution time values that can represent execution time of functions, loops, specific block of code or the whole application.

Generally, this approach provides summaries of accumulated times. The time is aggregated to have an idea about the execution time of requested application parts. Aggregate system timing generates data that can identify where a system spends the majority of its time, but not when and why. This method is still simple and fast to get preliminary performance data. However, its use for large applications seems to be irrelevant because of the enormous amount of instrumentation if every instruction should be timed. An example tool is PTR that stands for Portable Timing Routines [L1]. This is a project of the Parallel Tools Consortium (Ptools) [L2] that provides efficient timers for many platforms via a standardized library interface.

2.2.2. Profiling

Profiling provides an easy mechanism to collect reduced set of performance data. Generally, this technique serves for getting accumulated values of specific part of code. Typically, it is used to measure the number of times a given application part such as function, loop or code block is invoked. In this approach the profiling library contains implementation of the functions that can summarize the application execution and then calls to these functions must be inserted into the application code. Profiling provides a user with a kind of report about the application execution. It is not a way to locate exactly the performance problem, but it gives a general description of the application behavior in different categories. For example, it is possible to indicate functions that get a dominating percentage of the application execution.

There are two ways to provide profiling:

- Counting – it records the number of times an event occurred, but not where or why. Given both counts and total times, it is possible to accurately compute average execution times.
- Sampling – it allows one to obtain periodically the system state and increment a counter that corresponds to the observed state. Standard profiles sample the program at fixed time intervals. At each of these intervals program execution is interrupted and certain measures are taken and accumulated in the table. The produced histogram or table is called the execution profile. Typically, sampling provides information about how much CPU time is used by each function or subroutine in a program.

Some example profiling tools are:

- PAT – Performance Analysis Tool [Gal98] – it is a profiling tool developed by Silicon Graphics/Cray Research [L3, L4]. It uses sampling and accesses to hardware performance information to obtain an execution time profile for application functions.

- Apprentice [Gal98] – it is a successor of PAT profiling tool which usage and information are more complex than in PAT. It uses source code instrumentation through compiler switches and provides statistics on the level of functions and basic blocks.
- Xprofiler [L5] – this is the X-Windows Performance Profiler developed by IBM corporation [L6] that uses procedure-profiling information to construct a graphical display of the functions within the application. Xprofiler provides identification of the functions that are the most CPU-intensive.

2.2.3. Event tracing

This technique generates a sequence of event records. Each event is some significant activity and is an encoded instance of the action and its attributes. Typical record includes what happened in the application, when, where, and in which circumstances. It may contain information about what action occurred (e.g. what function was invoked), a timestamp, a place in the application (e.g. in which place of the invoked function, source code line number) and execution details (such as machine name, process identifier, function parameters and circumstantial parameters).

Typically in the classical analysis, the parallel program is executed under control of a monitoring tool that generates a trace file. The main purpose of the trace file is to illustrate the behavior of the program. Therefore, this trace file includes all the events recorded during the execution of the application. To generate such a file instrumentation must be inserted into the application at each necessary point. This instrumentation gathers all required performance data, creates an event record and allows for saving just created event to the file.

One of the possible alternative to the trace file is sending event records directly for the performance analysis purposes. In this sense, the analysis can be performed dynamically during the application execution and there is no need for saving information on the disk. We will explain this kind of performance analysis in Chapter 3.

Once the application has been terminated, the monitoring tool provides the user with a set of trace files. Generally, there is a distinct trace file saved locally. One trace file can be created for each machine (or processor or process – it depends on the implementation of the monitoring tool). Therefore, all events from all trace files must be merged into one global trace file. However, in this point a parallel distributed application cause an important problem, namely clock differences of a set of machines. Each event contains a timestamp that may be generated concerning time on a local machine. If clocks are different on different machines, event timestamps will also differ. In this situation, there is a need for timestamp adjustment, because only ordered events can be analyzed correctly.

Event tracing provides a good base for the application performance analysis. This technique is the most invasive technique, but it is also the most general and the most flexible. The main advantage is the big quantity of information about the application execution. A generated global trace file is representative of what really happened in the application, hence it allows for the reconstruction of the application execution. The disadvantage of tracing is its potential intrusion, the implementation complexity and large amount of produced data. The big information quantity causes the storage problem. A trace file containing events from an application that has been executed for many hours or even days, may occupy huge amount of disk space. Therefore, some precautions are taken into account by the developers of tools that monitor the application using tracing method. To reduce the amount of generated data, these tools provide for example selective instrumentation or binary/compressed trace file format. The well known examples of tools that base the monitoring phase on the event tracing are: PICL, Tape/PVM and VampirTrace. We explain them later in this chapter.

2.2.4. Hardware counters

The monitoring type described above is characteristic for the software performance monitoring. The application source code is changed in order to obtain information about different software segments (e.g. functions, statements). However, we can also distinguish a second type of monitoring namely hardware performance monitoring (HPM) [L7]. HPM provides statistics of the hardware operations performed by CPU. Generally such a monitor is based on the counters and contains a small set of registers that count events, which are occurrences of specific signals related to the processor’s functions. The example operations are: floating point operations (multiply, add, multiply-add, divide, etc.), integer operations or memory operations. Monitoring these events facilitates correlation between the structure of the source code and the efficiency of the mapping of that code to the underlying architecture. Software and hardware performance monitors provide complementary information.

Some example tools are:

- PAPI – it stands for Parallel Application Program Interface. This is a standard API for obtaining the values of hardware counters available on modern microprocessors [L8]. This project forms a part of the working group of a Ptools Consortium.
- PCL – it is Performance Counter Library [L9] developed Research Centre Juelich in Germany [L10]. This is a portable API for accessing hardware counters that provides interface for many languages.

2.3. Visualization

Once the application has been instrumented and performance data is available, the second step for the measure and modify approach can be performed. This step visualizes the generated trace file. Tools that support this measure and modify method of the classical analysis can display post-mortem - after the execution of the program – the trace file. They do this usually via different perspectives such as gantt charts, bar charts or pie charts. Most of the visualization tools use detailed graphics to show the application execution. The displayed information may contain message-passing, collective communication, execution of application subroutines, and so on. The next step requires developers (or experts) to analyze illustrated information and detect potential problems. Then they must find causes that made the bottleneck problems occurred. In the following step a developer has to manually relate detected problems to the source code. The last step is to tune the application – fix found problems by changing the source code of the program. Then the modified program must be re-compiled, re-linked and restarted. Figure 2.2 shows the general view of the classical approach to performance analysis. There are many visualization tools that try to help the users by providing them with different views of the application execution by analyzing gathered trace files. We present the most known examples in Section 2.4. They offer fairly evolved interfaces and allow the user to navigate amongst the different views providing quite intuitive screens. With these tools, it is reasonably easy to see the general behavior of the application.

Although this approach has been used for many years, it still has several drawbacks:

- In general terms, it is a very time consuming task that can significantly delay the application deployment. The degree of expertise required to carry out this task is very high, especially the phase of relating performance bottlenecks to the source code of the application and deciding how the application should be optimized.
- Usually, large applications running for several hours produce a huge amount of data in the trace file that is difficult to manage and analyze. Additionally, to collect all the required information for the analysis, it may be necessary to heavily instrument the application. This instrumentation can provoke a significant level of intrusion and affect the real performance of the application.

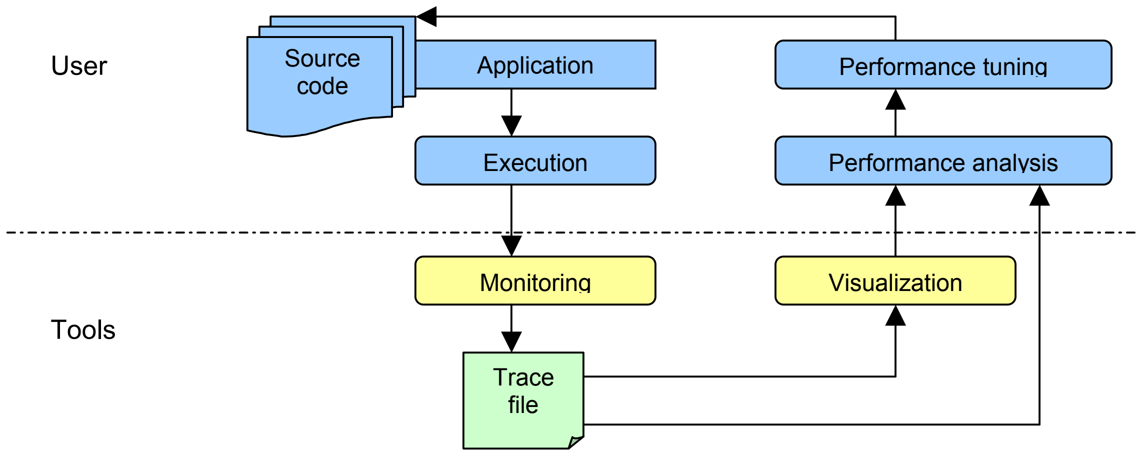


Fig. 2.2. The general view of the classical approach to performance analysis.

- Visualization tools do not scale very well. When the number of processes involved is high or the execution time is long it is difficult to have a clear picture of the behavior of the application, since the visualizations become unreadable.
- The analysis is based on a single execution of the application. It is suitable for stable applications that present the same behavior for different input data sets. However, when the application behavior depends on the input data, the modifications based on analysis for one particular execution can be inadequate for another execution.
- When the application behavior varies during the execution from one iteration to another (for example, a different number of null elements in a matrix), the useful modifications for one particular region of the application can be contradictory with the modifications required by other iterations. The problem is that there is a single copy of the source code and different iterations require different implementations.

- If the target platform is changed (number of processors, processor speed, network bandwidth, etc.) the required optimizations may be different. This becomes a significant problem in grid systems that exhibit highly dynamic and unexpected behavior.

Taking all these facts into account, the classical "measure and modify" approach based on visualization tools is only feasible for a reduced set of applications. Such applications cannot be very large, must have quite a stable behavior and cannot require a significant amount of instrumentation.

2.4. Example tools

Many monitoring and visualization tools, which try to help the user in the complicated application performance analysis and improvement, are available. There are some tools that can only generate trace files, some tools that can visualize them and the other tools that can do both things together. In the next subsections we present the most popular tools in the classical analysis area for monitoring and visualization purposes.

2.4.1. Tape/PVM

Tape/PVM [Mai95] was developed at LMC-IMAG Laboratory and it is one of the well known monitoring tools that generates trace files of PVM applications. It also provides the following utilities: library of C functions to easily read the generated traces and tool to transform the traces to PICL format. Therefore, Tape/PVM is a good base for other tools that can visualize an application execution and provide post-mortem performance analysis. It is focused on minimal overhead introduced into traced programs and causally coherent event dating using clock synchronization.

This tool uses a special preprocessing phase to insert instrumentation into a source code. The preprocessing phase invoked by the user consists in inserting a call to the Tape/PVM initialization function and in intercepting calls to the PVM library. For each PVM function there is an associated intercepting function which records the trace information before passing control to the actual PVM function. After this phase, the modified code must be compiled and linked with the Tape/PVM library by the user. Once the instrumented program has been launched, the first clock synchronization phase starts to collect differences of clocks from all machines. Once it is finished, a program starts its normal execution. Each process of a running instrumented program generates locally its own trace file. After execution of the program, TAPE/PVM joins all files into one global trace file transforming local timestamps of events into global timestamps. Therefore, events from different machines have the same global time reference, are comparable, and causally coherent.

Overhead introduced into an application due to the use of a monitoring tool can be significant, hence Tape/PVM limits the intrusion using appropriate techniques. It compacts the events, reduces the number of exchanged messages, and performs all additional tasks (e.g. clock synchronization, global time reference) before/after a program execution.

2.4.2. PICL

PICL – Portable Instrumented Communication Library [Gei90, L11] is a tool developed at Oak Ridge National Laboratory [L12]. PICL serves for portability, ease of programming and execution tracing in parallel program. PICL is a subroutine library that can be used to develop PVM-based programs that are portable across several platforms. It supplies low-level communication primitives. However, it also simplifies parallel programming by providing a set of high-level communication routines.

The library has a built-in mechanism of trace file generation. Using special routines provided by PICL library, it is possible to invoke the tracing of processes, as well as control the type and amount of tracing data. Including such tracing routines' calls in the application code, a user requests the PICL library to activate routines that produce time stamped records and generate a trace file on each processor. Although the set of tracing routines provided by the library is small and easy to use, the recompilation and linking phase is required. The user has a possibility to configure tracing data. He/she can choose what type of trace records (e.g. user-defined, event, statistics) and what type of event (e.g. user-defined, communication, I/O, synchronization, resource allocation) is to be generated. Therefore, the number of traced data can be reduced if necessary. The trace file contains one record per line, and each record comprises a set of fields that specifies the record type, event type, timestamp, processor id, process id, number and description of other data fields that are common for a particular event. After the application execution, all generated trace files are collected and sorted by time.

The PICL library has been evolved to MPICL that provides a mechanism for collecting information from MPI-based programs.

2.4.3. ParaGraph

ParaGraph [Hea95, L13] is the visualization tool developed at University of Illinois for the trace files generated by the MPICL library. It provides the dynamic, graphical, and detailed animation of the behavior of message-passing based programs (MPI), as well as summaries of overall program performance. It replays in a visual form events that happened during parallel program execution. ParaGraph provides to a user a set of views that are divided into three categories: processor utilization, communication between processes and task information. The example view of the process utilization is showed in Figure 2.3. The utilization view displays the total number of processors in each of three states (busy, overhead and idle).

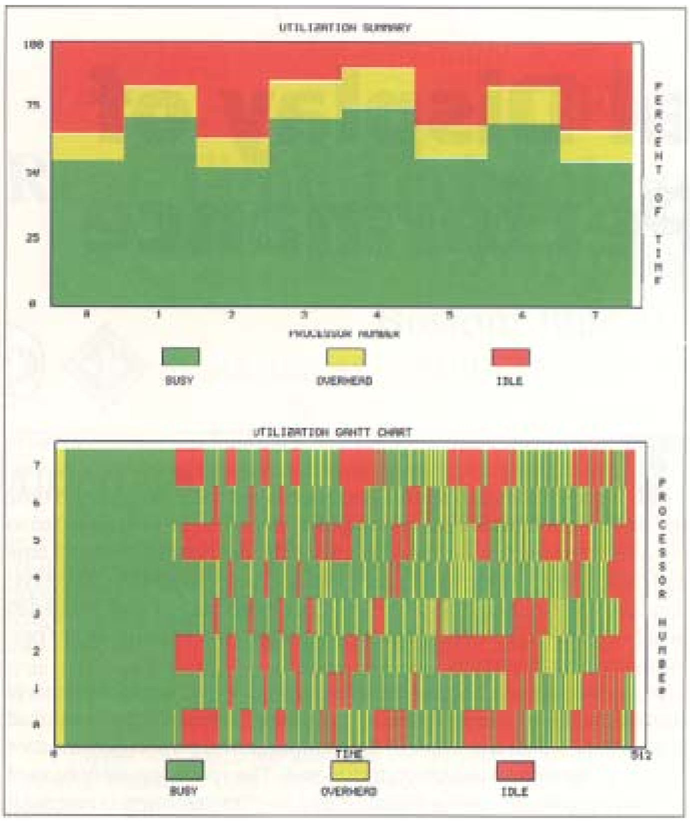
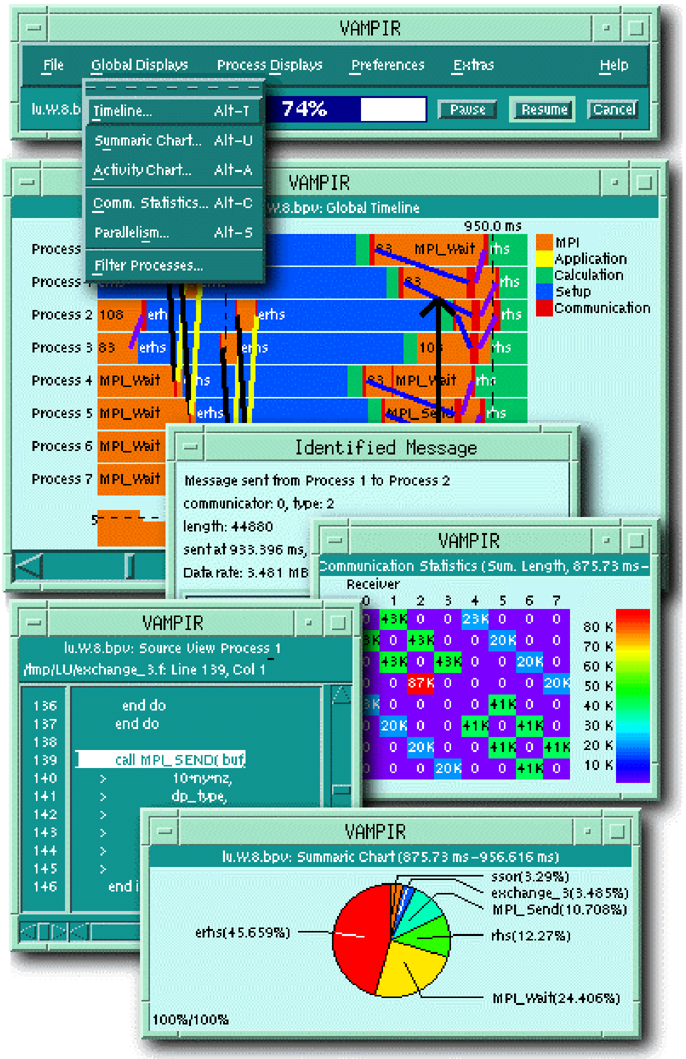


Fig. 2.3. View of processor utilization summary and utilization Gantt chart provided by Paragraph.

Although ParaGraph takes as an input trace files in the MPICL format, the tool depends only on input data. Therefore, it can perform equally well any trace file that has the same format and semantics as PICL/MPICL. If other message-passing application is instrumented and generates trace files in PICL format, ParaGraph can process them showing the program behavior.

2.4.4. Vampir

The VAMPIR [Nag96, L14] (Visualization and Analysis of MPI pRograms) is distributed by Pallas company [L15], a member of the ExperTeam group. This is the commercial visualization tool especially for MPI-based applications. It provides a variety of graphical displays that present important aspects of the application behavior (see Figure 2.4). It also supplies flexible filter operations to reduce the amount of information to be displayed and forward/backward motion in time. Vampir supports evaluation of load balancing, analysis of performance of subroutines or code blocks, and identification of communication bottlenecks. The tool displays information about communication patterns, parameters and performance.



```
Message sent from Process 1 to Process 2
communicator: 0, type: 2
length: 44880
sent at 933.396 ms,
Data rate: 3.481 MB
```

0	1	2	3	4	5	6	7
0	43K	0	0	23K	0	0	0
3K	0	43K	0	0	20K	0	0
0	43K	0	43K	0	0	20K	0
0	0	87K	0	0	0	0	20K
3K	0	0	0	0	41K	0	0
0	20K	0	0	41K	0	41K	0
0	0	20K	0	0	41K	0	41K
0	0	0	20K	0	0	41K	0

```
       end do
      end do

      call MPI_SEND( buf,
     >            10*ny*nz,
     >            dp_type,
     >
     >
     >
     >
     end i
```

Fig. 2.4. Different views of the application behavior displayed by Vampir.

As all mentioned tools, Vampir also uses trace files to visualize behavior of an application after its execution. This tool has its own mechanism to monitor an application, namely VAMPIRtrace. It is a library for tracing operations of MPI-based applications. It uses the profiling interface of MPI library to record point to point communications, collective operations, MPI I/O operations and user-defined procedures. Generally, the trace file generation is convenient, because a user does not need to change the source code, only to link the application with the VampirTrace library. However, when the user-defined procedure is to be traced, the VampirTrace API must be used to modify the application and hence the recompilation phase is required. The tool also provides event filtering in a form of configuration file read before the application execution. This feature allows for collecting only selective events, what can significantly reduce the size of trace file.

2.4.5. Pablo

The Pablo [Ree93] environment developed at University of Illinois includes application performance instrumentation, graphical and sonic representation of the collected data and the data amount reduction. It provides a manual application instrumentation as well as a graphical interface for interactive specification of instrumentation points. Pablo allows for capturing procedure calls, inter-process communication, and input/output operations.

Pablo offers many different graphics (that can include also sound) to the user where performance data is constructed as analysis graph [L16]. The example views are presented in Figure 2.5. The resulting performance analysis graph can be saved in a compact, portable and extendable format. In addition to graph-based analysis, Pablo contains a statistical analysis software that can compute and display histograms of specific data values. Moreover, this tool supports the captured data visualization by means of a virtual world. Pablo has implemented performance data presentation metaphor that shows the application execution in the three-dimensional space.

The instrumentation software supports tracing, interval timing, and counting. The tools generates its own trace files which contain a set of event in a typical event format (what happened, when and where). As majority of the mentioned tools, the Pablo trace library has been also designed to reduce perturbations caused by monitoring and dynamically altering the number of data to be traced. If the overhead exceeds specified thresholds, the instrumentation software will automatically convert more invasive instrumentation e.g., event tracing to a lower one e.g., counting.

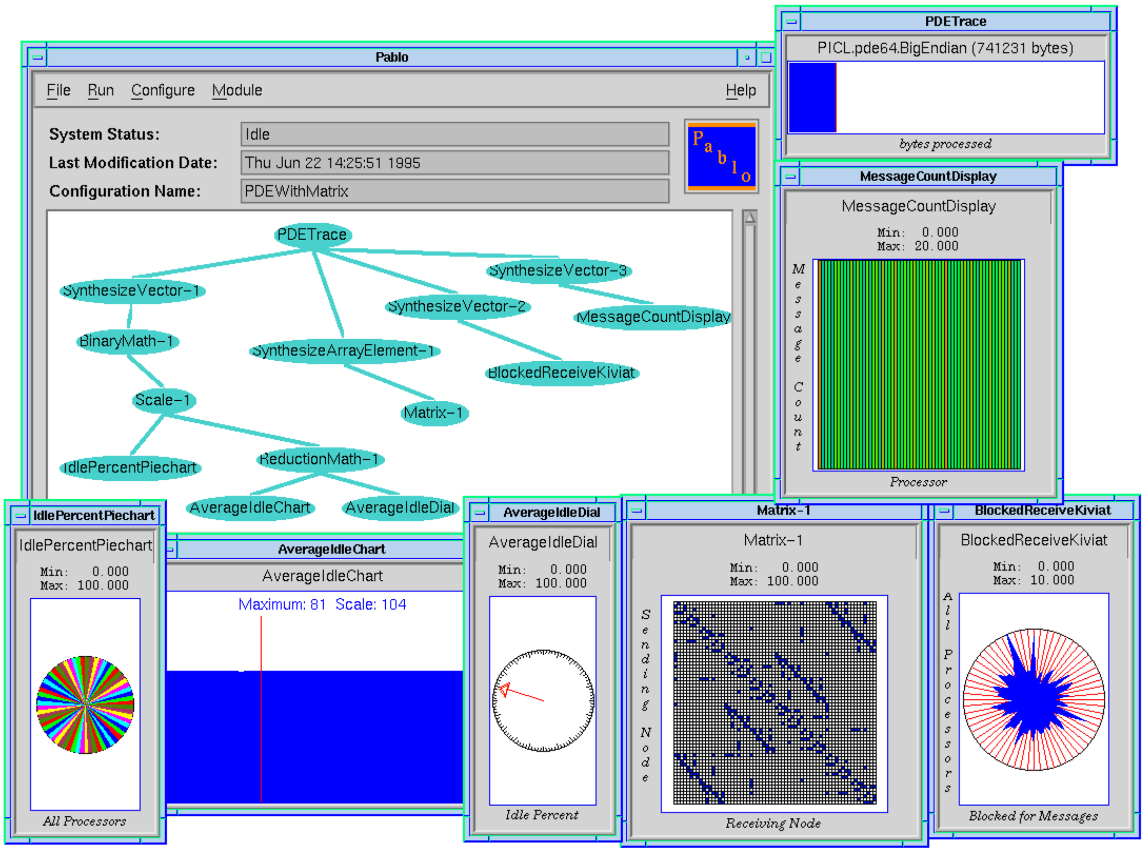
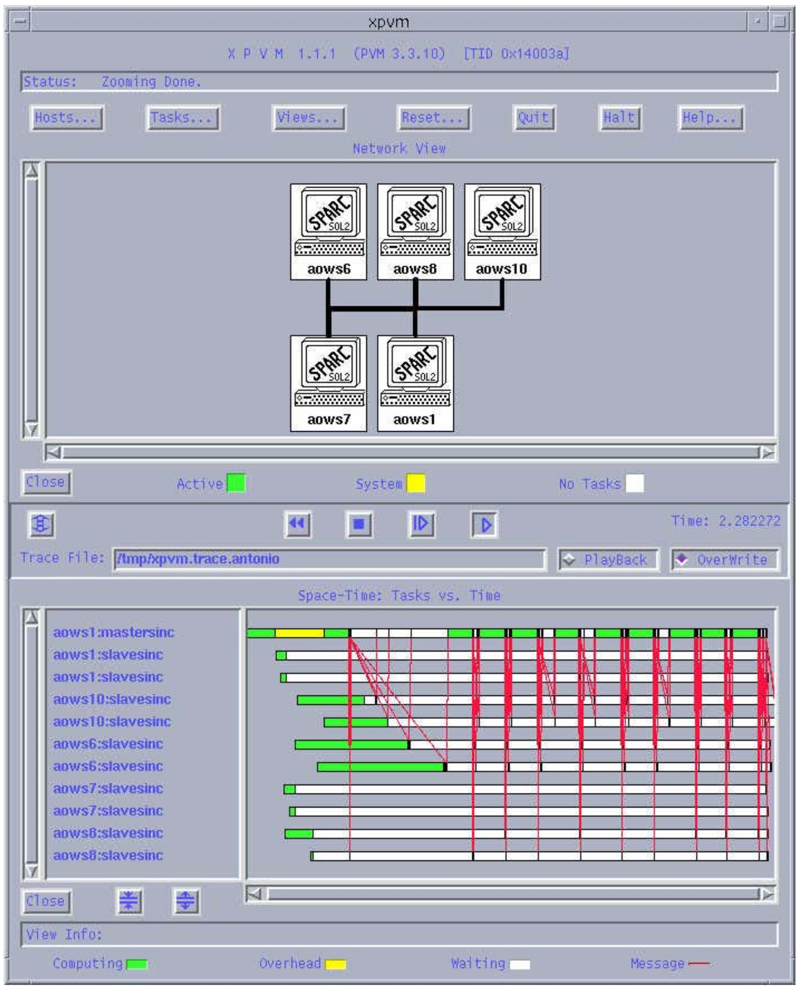


Fig. 2.5. Different views of the application execution provided by Pablo toolkit.

2.4.6. XPVM

XPVM [Gei94] is a graphical console and monitor for PVM-based programs developed at Oak Ridge National Laboratory. It is usually integrated with the PVM library. XPVM can be used as a graphical environment to manage virtual machine and parallel application. It provides creation and elimination of tasks and machines.

XPVM serves also as a monitoring tool. It uses the event collection mechanism integrated in PVM. During execution of an instrumented program, PVM kernel routes events to XPVM. XPVM has its own visualization utility, presented in Figure 2.6, hence it is able to perform received information on-line and display it in “real time”. A very useful feature of the tool is the ability to show the communication between processes and the group operations like barrier. XPVM main window contains two parts: configuration of the virtual machine and the state of the application tasks. XPVM provides also possibility to save events of program execution to a trace file. This trace file can be used as input to the visualization process. XPVM may be also used for post-mortem performance analysis.


X P V M 1.1.1 (PVM 3.3.10) [TID 0x14003a]

Status: Zooming Done.

Time: 2.282272

Trace File: /tmp/xpvm.trace.antonio

Fig. 2.6. Main window of XPVM that displays virtual machine and application behavior.

2.5. Predictive analysis

A prediction or forecast is the result of an attempt to produce a most likely description or estimate of the actual evolution of a variable/system in the future. Predictive analysis of application performance is based on the analytical models that are constructed in order to predict future application behavior for different conditions. Moreover, these models provide the programmers with a feasible expression of the program execution so it is easier to draw conclusions about the program performance. The advantage of modeling is that it enables the prediction of application performance for different input data, machines and problem definition. However, the definition of accurate model is complex and requires expert knowledge. To facilitate this task, predictive tools avoid the process of understanding the performance details by abstracting them in higher level expressions that are useful for the programmers. In this approach the model construction is automated and as an output the user obtains the application behavior model.

In the following sections we present example tools that are based on predictive analysis approach.

2.5.1. Lost Cycles Analysis

Lost Cycles Analysis (LCA) [Cro93] is a toolkit for the performance analysis of the parallel applications that was developed at the University of Rochester. The tool automates the performance model construction as a function of runtime factors. The analysis is divided in two phases: predicate profiling of the application to obtain measurements and modeling of those measurements.

During the predicate profiling phase, the empirical measurements must be obtained to model the application behavior. The measurements are the lost cycles associated with overheads. Overheads are execution delays expressed in seconds and they are divided into categories that depend on the source of the wasted time (e.g. time spent in tasks not directly related to the computation such as communication or synchronization, delays caused by load imbalance). The overhead categories are defined via performance predicates, which are logical statements that represent the occurrence conditions of the categories. To perform the profiling, first, the user defines the environment variables that will affect the application execution. The user samples the valid ranges for each variable and the combination of the sampled values defines the parameters that are used in the experiments. Typically these parameters are: the number of processors used, the number of iterations performed by a specific part of code or data structure dimension. The profiling phase ends with execution of the experiments and data collection. In the LCA this phase is performed by means of the performance profiler (pp tool).

The modeling phase assigns models to overhead categories. These models are functions of environment variables that rule the application's execution. The determined models are fittings of the measured data to default models. The modeling contains two steps: one-dimensional and multi-dimensional model generation. In the first step user generates models for each environment variable and each overhead separately (e.g. expression that models load imbalance in function of number of processors and data size). Next the tool called lca uses the measurements performed in the profiling phase to parameterize the models finding the appropriate coefficients that approximate the reality. The resulting models for each category of overhead capture the effects of varying environment parameters in isolation. In the second step the user combines one-dimensional models into multi-dimensional models with a help with the lca tool. To produce the final performance function, the user must add or multiply models of the overhead categories.

The objective of the toolkit is to provide a mathematical representation of the performance problems – overheads – found in the application execution. The advantage of such a representation is high level view of the execution details. However, the difficulties rely on the user that must manage the construction of the application models that accurately reflect the reality.

2.5.2. P3T

P3T, a Parameter-based Performance Prediction Tool [Fah93], is an interactive performance estimator that helps users tune scientific Fortran programs. This tool was developed in the context of the Vienna Fortran Compilation System (VFCS) [Ben95] which enables the estimator to exploit considerable knowledge about the compiler's analysis information and code restructuring strategies. The VFCS translates Fortran programs into explicitly parallel message-passing programs and P3T guides the interactive and automatic restructuring of programs under this system. P3T detects bottlenecks in the program, identifies the causes of performance problems, and guides users in selecting effective program transformations to gain performance. The tool focuses on the following aspects of parallel programs: load balance, data locality, communication, and computation overhead. As an input P3T takes a list of performance parameters that are estimated at compilation time. The example parameters are: number of transfers, amount of data transferred, transfer time, computation time, work distribution and so on. These parameter provide the required abstraction to express the performance application quality. All parameters can be assigned to blocks of code such as loops or procedures.

The performance analysis is guided via GUI that directs the user to the computational bottlenecks that prevent the program from performing well. In addition, P3T allows performance data to be filtered and visualized at various levels of detail.

2.5.3. Dimemas

Dimemas [L17] is a tool developed by CEPBA center of Universitat Politècnica de Catalunya [L18]. This is a simulation tool for the parametric analysis of the behavior of MPI applications. It enables the user to develop and tune parallel applications on a workstation, while providing an accurate prediction of their performance on the parallel target machine.

First the application must be instrumented in order to obtain trace file. Dimemas instrumentation library is based on the profiling VAMPIRTrace tool. This library usually records the CPU time consumption, communication primitives and event information such as function begin/end, value of variables, value for internal processor registers. Moreover, using GUI a user can model the architectural parameters of machines. The simulator allows specifying different task to node mappings.

With the records read from the trace files and specified architectural parameters, Dimemas will rebuild the time behavior of a parallel application. The tool simulates the application execution scaling the time spent in each block according to target CPU speed. As an output Dimemas generates trace files that are suitable for the visualization purposes and in particular for Vampir tool. The results include global application information: execution time and speedup. Additionally, for each process the tool gives information about execution time, blocking time, computation time, number of messages sent and received, volume of communication data. Moreover, Dimemas is able to determine the critical path which returns the longest communication path of the application.

2.6. Conclusions

As we have seen, the classical analysis tools support a user in performing two phases: monitoring and then visualization or prediction of the application behavior. The first phase is facilitated by monitoring tools that are generally based on the use of the tracing method. The tool that uses this technique to monitor the application, inserts instrumentation into the important parts of the application code and generates trace files. Once a trace file has been generated, the second phase is performed by visualization or predictive tools.

Many monitoring and visualization or predictive tools, which try to help a user in the complicated application performance analysis and improvement, are available. There are some tools that can only generate trace files, some tools that can visualize them or predict the behavior and the other tools that can do both phases together. Nevertheless, the tools that support the classical approach are very similar in their purposes and results provided to the user. Using this kind of tools, the user must have a great knowledge of parallel applications and experience in developing them in order to improve their behavior.

Chapter 3
Automatic performance analysis

This chapter presents the general idea of the automatic performance analysis of parallel and distributed applications. Then, we introduce an extension to the automatic approach, namely dynamic performance analysis, that performs the application analysis automatically during run time. Finally, we are going to present examples of tools that support both automatic and dynamic performance analysis of parallel programs.

3.1. Overview

Although visualization tools based on the traditional approach are often very helpful, developer must have a great knowledge of parallel systems and experience in performance analysis in order to improve the application behavior. To overcome difficulties of the "classical approach" it would be very important to offer to the users tools that would guide them in the tuning process. It would allow them to avoid the degree of expertise required by the visualization tools. Such tools should introduce some automatic features. Carrying out some steps of the application performance analysis automatically, the participation of the user could be reduced.

To decrease developers' efforts, especially to relieve them of duties such as analysis of graphical information and determination of performance problems, an automatic parallel program analysis has been proposed. Tools using this type of analysis are based on two principles. First, they use collection of measurements gathered from the application execution and provided by a monitoring tool. The application is instrumented before it is put into the execution and instrumentation is inserted into all necessary points. Second, they are based on the knowledge of performance problems. Once measurements have been collected, the automatic analysis process can be performed. This process is a search for performance problems within the information obtained from the execution. The principal question here is how to detect performance bottlenecks. Experience with parallel applications has shown that many of them have well-recognized performance problems. To search for a performance problem, a tool should be then supported with the information about possible bottlenecks and how to find them. The objective is to comprise information about both application and parallel system features. Potential bottlenecks can be represented as a knowledge provided to a tool. Such a knowledge may contain performance models that provide a way for understanding performance problems. Using a good analytical model, the application behavior might be predicted, bottlenecks can be found, as well as their causes and optimizations can be deduced and provided to the user. Unfortunately, the creation of models is not an easy task and usually it is a compromise between simplicity and accuracy.

Each tool then has built-in performance model for typical kinds of applications, hence it is able to identify critical bottlenecks and help in optimizing applications by automatically giving suggestions to developers. These hints or recommendations expose the performance problems showing the parts of the application, which are performing poorly. Figure 3.1 shows the general parts of automatic analysis tool that support a user in analyzing and improving application performance. There are several tools available that support this approach such as KappaPi, Paradise or AIMS. We will talk about them later in this chapter.

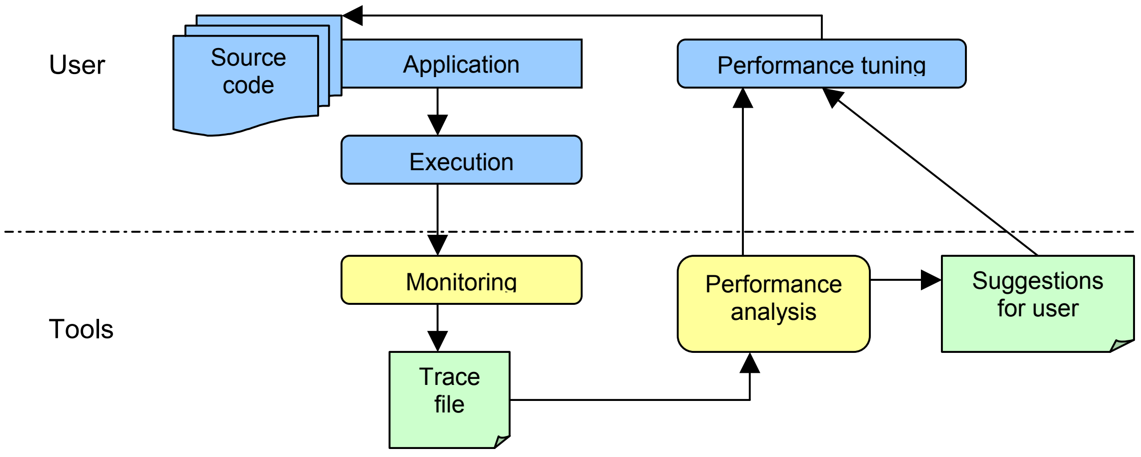


Fig. 3.1. The general view of the automatic approach to performance analysis.

In this approach the performance bottleneck search is still based on trace files. This automatic analysis can be called static since is done post mortem – after the application finishes its execution. The visualizations are replaced with automatic analysis and direct recommendations about detected problems. These tools significantly reduce the amount of time spent by developers in performance analysis, since they are supported with more automation in the whole tuning process and receive information that is more accurate. Generally, given hints are representative and useful for only one application behavior. The tool analyzes a trace file generated for one application execution. If the application is executed again and has different input data or changes the behavior during the execution, the previously performed analysis can be inadequate. Therefore, tools are more adequate for the same input data and stable applications.

3.2. Dynamic performance analysis

Although analysis is facilitated in the presented automatic analysis approach, the developer must manually perform the application tuning. In addition, some of the drawbacks mentioned above with respect to the visualization approach are still present in this automatic performance analysis:

- Fully instrumented application.
- Trace file based analysis.
- Single run of the application in a given environment.
- Stable behavior required.

Therefore, the automatic analysis approach was extended from the static version to the dynamic automatic analysis. In this case, performance analysis is done on the fly, during the execution of the application in a fully automatic manner and avoid the need for the manual instrumentation.. It implies the necessity for the on line monitoring, where the principal advantage is that any trace file is no more needed for analysis. Figure 3.2 presents basic view of the dynamic performance analysis.

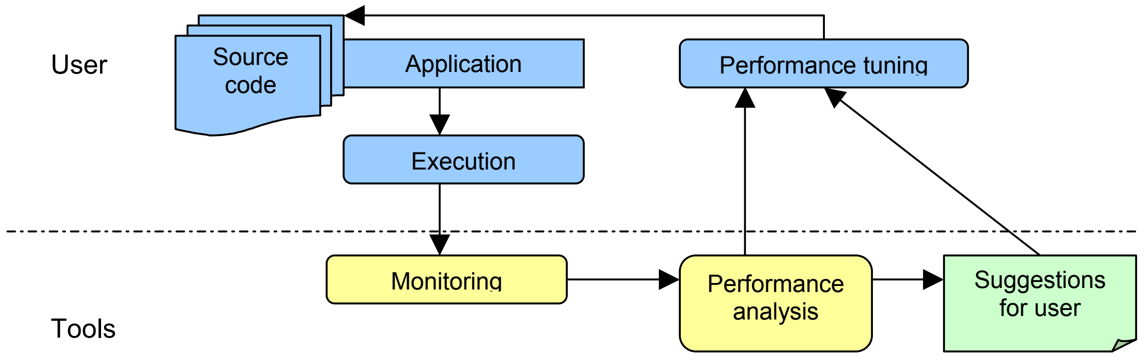


Fig. 3.2. Basic scheme of the dynamic and automatic performance analysis.

This approach allows for control of the amount of instrumentation inserted in the application by applying dynamic instrumentation techniques. The monitoring can start with a very simple instrumentation and when some particular conditions are detected, the additional instrumentation can be introduced. When the conditions disappear, it is possible to eliminate the extra instrumentation. In this approach, the analysis must be done during the execution of the application what implies some extra overhead. Therefore, the analysis must be relatively simple so as to reduce the overhead as much as possible. The principal and most known tool that provides dynamic performance analysis is Paradyn. We present it later in this chapter (see Section 3.3.4).

By using a dynamic analysis tool, the problems can be identified significantly faster than in a post-mortem approach. The dynamic approach is best suited for iterative programs and can handle long-running applications with high data volumes. However, to solve the detected problems, it is necessary to stop the application, modify, recompile and rerun it. This implies that the work carried out is aborted and a new execution is launched. Similarly to static analysis, the dynamic analysis is based on a single run of the application. When the application behavior depends on the input data or on the iteration of the execution, the suggested recommendations can be inadequate for a further run of the application.

3.3. Example tools

In the next subsections we present tools that support the user with automatic and dynamic analysis during the application performance improvement. All these tools have provided successful results.

3.3.1. KappaPi

KappaPi [Esp98] stands for a Knowledge-based Automatic Parallel Program Analyzer for Performance Improvement. This tool was developed at Universitat Autònoma de Barcelona. KappaPi is a static automatic performance analysis tool based on a trace file post-mortem analysis and a knowledge base that includes the main bottlenecks found in message passing applications. The tool helps the user in the performance improvement process by detecting the main performance bottlenecks, analyzing the causes of those problems, and relating the causes to the source code of the application. KappaPi provides also some suggestions about the detected bottlenecks and the way to avoid them.

In the preliminary step the application must be executed with Tape/PVM monitoring tool in order to get the trace file that will be then analyzed by the KappaPi analyzer [Esp00].

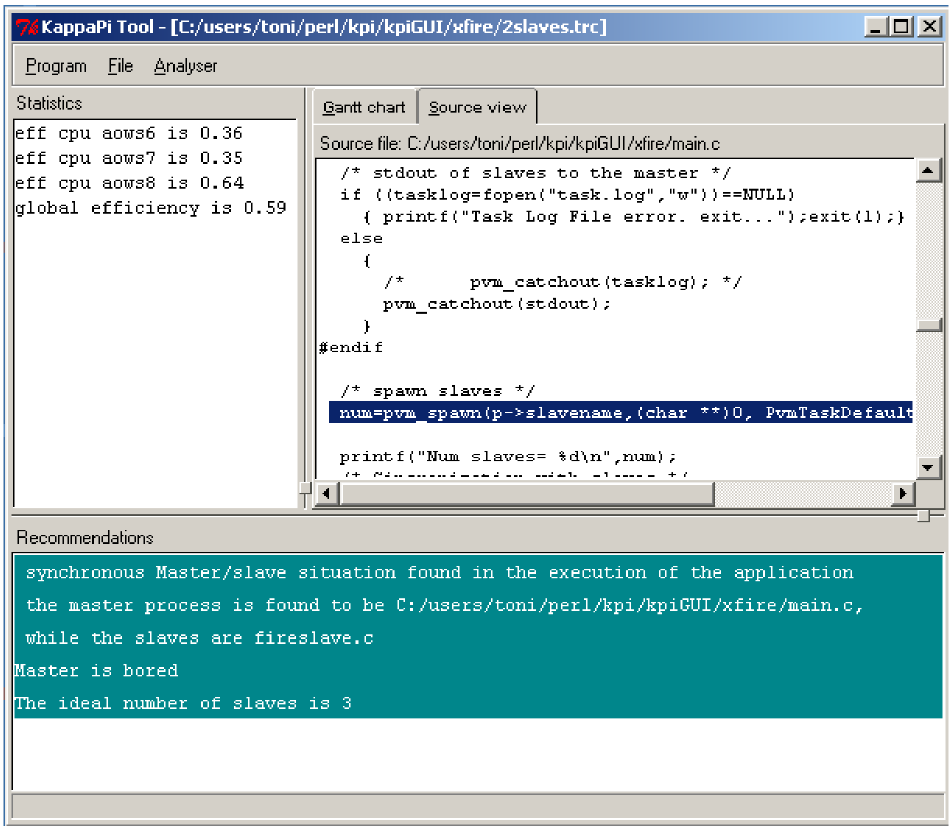
Once the trace has been generated, KappaPi tool can be invoked. In a first step, KappaPi makes a general overview of the application performance by measuring the efficiency of the different processors of the system. KappaPi considers as performance inefficiencies those intervals where processors are not doing any useful work; they are just blocked waiting for some message. So, the efficiency of a processor is considered as the percentage of time where it is doing useful work. When there are idle time intervals, these time intervals should be avoided in order to improve the performance of the application. The best situation would be to have all the processors completely busy doing useful work during the execution of the application. In this first step the user gets some information about the overall behavior of the application, but without any idea about the bottlenecks and their causes.

After this initial classification, KappaPi starts the deep analysis looking for performance bottlenecks. KappaPi takes chunks from the trace file and classifies the performance inefficiencies detected in that chunk. It must be pointed that several inefficiencies can correspond to the same performance bottleneck, because in many cases the inefficiencies are repeated along the execution of the application. The detected bottlenecks are classified in a table according to the inefficiency time incurred. After analyzing the first chunk, the second chunk is analyzed and a new table is built and joint to the initial one in such a way that the new inefficiency time of the same bottleneck is added to the first one. The process is repeated for all the chunks and finally KappaPi provides a sorted table with the worst performance bottlenecks.

The next stage in the KappaPi analysis is the classification of the most important inefficiencies. For that purpose, the tool relates these inefficiencies with some existing behavior categories using a rule-based knowledge system. From this point, the inefficiencies are transformed into specific performance problems that must be studied to build up some hints to the users. To carry out this classification, KappaPi tool takes the trace file events as input and applies the set of rules deducing a list of behaviors. The initial list of events is the starting point for the detection algorithm. These events are the base for the detection of higher order facts. The just deduced facts are kept in a list (accumulated list) so that, in the next iteration of the algorithm, higher order rules apply to them. The process will finish when no more facts are deduced.

After the performance problem has been identified when fitting in one of the categories of the rule-based system the query process is finished. The next step in the analysis is to take advantage of the problem type information to carry out a deeper analysis that determines the causes of the performance bottleneck with the objective of building an explanation of this problem to the user.

Figure 3.3 represents the KappaPi graphical user interface. The main program window provides to the user very useful information. It contains the following parts: statistics with general list of efficiency values per processor, hints about the actual quality of the application performance together with recommendations about what changes can be applied in order to improve the performance, source code view with highlighted critical lines and Gantt chart representing execution visualization.


KappaPi Tool - [C:/users/toni/perl/kpi/kpiGUI/xfire/2slaves.trc]
Program
File
Analyser
Statistics
Gantt chart
Source view
Source file: C:/users/toni/perl/kpi/kpiGUI/xfire/main.c
Recommendations

```
eff cpu aows6 is 0.36
eff cpu aows7 is 0.35
eff cpu aows8 is 0.64
global efficiency is 0.59
```

```
  /* stdout of slaves to the master */
  if ((tasklog=fopen("task.log","w"))==NULL)
    { printf("Task Log File error. exit...");exit(1);}
  else
    {
      /*       pvm_catchout(tasklog); */
      pvm_catchout(stdout);
    }
#endif

  /* spawn slaves */
  num=pvm_spawn(p->slavename,(char **)0, PvmTaskDefault

  printf("Num slaves= %d\n",num);
```

```
 synchronous Master/slave situation found in the execution of the application
 the master process is found to be C:/users/toni/perl/kpi/kpiGUI/xfire/main.c,
 while the slaves are fireslave.c
Master is bored
The ideal number of slaves is 3
```

Fig. 3.3. Final view provided by KappaPi when analyzing master/worker application.

3.3.2. Paradise

Paradise [Kri96] stands for PARallel programming ADvISEr. The tool was developed at University of Illinois and provides automation to the parallel application optimization using post mortem analysis. It not only finds performance bottlenecks, but also presents solutions to problems found. This is a framework that analyzes trace files building an event-graph representation of the program's execution. In this tool the behavior of applications and systems are modeled as a set of objects that have certain functionalities and interact with each other. These objects and their interactions can simulate the events that are generated during the application execution. Then the tool determines characteristics of this application and uses heuristics to find possible solutions to optimize application performance. Finally, Paradise generates hints saving them to a file. In general, Paradise represents similar approach to the one described in KappaPi. However, in comparison to KappaPi, Paradise is not able to combine performance problems with an application source code.

Paradise works in cooperation with a run time system that uses generated hints to parameterize optimizations and select between different alternate optimization strategies. The whole project is based on the parallel object-oriented language Charm++ [Kal93] which is an extension of C++. It enables the benefits of object-orientation to be applied to the problems of parallel programming. The basic work unit in Charm++ is a chare, which is a concurrent C++ object. A chare type is a C++ class that contains data and functions which may be triggered by the arrival of messages.

Once the application has been written in Charm++, it can be run and monitored to obtain trace files with necessary performance data. The application execution is represented as an event graph, which is a task graph constructed using generated trace files. The event graph constructed by Paradise consists of vertices representing entry-function executions, edges representing messages between entry functions and edges for dependences between methods (these dependences must be specified in the language or generated by the compiler). Analyzing trace files Paradise intents to discover the characteristics of the application by searching for common bottleneck patterns. The characteristics and possible optimizations researched in Paradise are for static and dynamic object placement, scheduling, granularity control and communication reduction. Once a bottleneck has been found, the tool gives suggestions to the user.

3.3.3. AIMS

AIMS [Yan96, L19] stands for an Automated Instrumentation and Monitoring System. This tool was developed at NASA AMES institute [L20]. The tool provides utilities of measurement and performance analysis for message-passing programs written in Intel's NX, PVM or TMC's CMMD communication libraries. AIMS consists of four main components: a source code instrumentor – which automatically inserts instrumentation into the application; a run-time performance-monitoring library which collects performance data; two tools that process the performance data – trace file animation and analysis toolkit; a trace post-processor that removes overhead introduced by monitor.

Instrumentor provides graphical user interface to insert instrumentation into subroutines invocations, synchronization operations and message-passing supporting different communication libraries. It also generates two key data structures: an application database that stores static information about the application source code (e.g. file names and line numbers of instrumented points) and enabling profile that contains information about inserted instrumentation. After the use of instrumentor, instrumented source code must be recompiled and linked with the monitoring library. The monitor reads the profile at the beginning of the execution and hence it can generate trace files. For each processing node, monitor writes events to the buffer. If the buffer is full or the application has finished, monitor flushes the buffer to the file. The buffer size can be controlled and configurable by the user.

After the application execution, trace file can be analyzed and displayed by the visualization toolkit (see Figure 3.4). AIMS provides a set of detailed and animated views

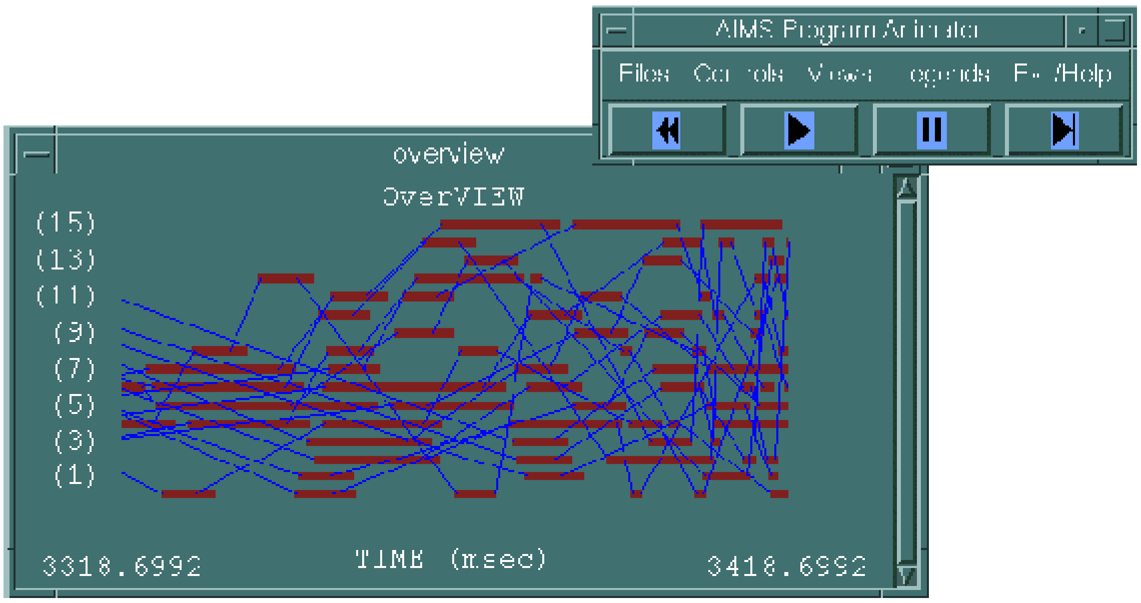


Fig. 3.4. Detailed execution analysis presented by AIMS visualization toolkit.

indicating certain constructions of the execution in time. It also collects and tabulates statistics that reflect the cumulative activity of the program. AIMS generates a list of resources-utilization statistics what can help a user to find inefficient code sections. The tool has also capability to map an event displayed on a view to the corresponding application source code. AIMS contains a trace file post-processor that removes as much intrusion as possible from a trace file.

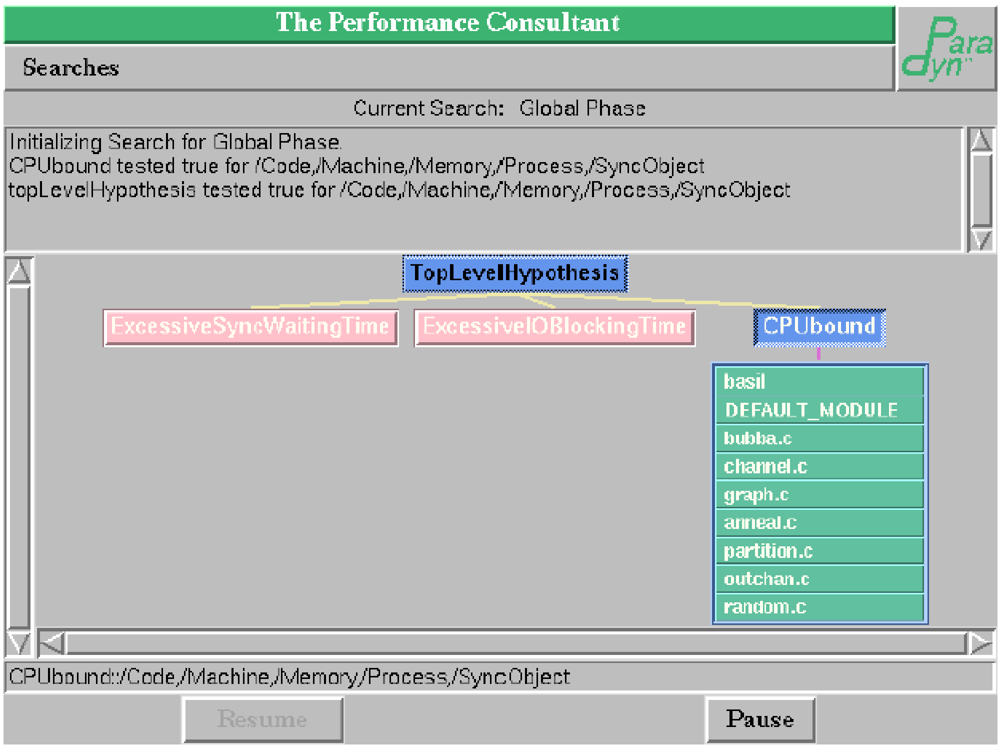
3.3.4. Paradyn

Paradyn [Mil95, Par03, L21] is a performance tool for large-scale parallel applications which was developed at University of Wisconsin [L22]. It provides monitoring and automatic analysis “on the fly”. This tool does not require any trace file of the application execution. It takes advantages of a special monitoring technique called dynamic instrumentation that defers instrumenting the program until it is in execution. Therefore, Paradyn is able to insert and modify instrumentation during run-time without any changes of the program source code.

Paradyn also provides automatic performance analysis of the running application. While the main objective is to do the monitoring and analysis phases during run-time, Paradyn is able to make decisions and give results dynamically. It automatically identifies these parts of the application that consume most resources. Paradyn searches for performance problems using the W^3 search model (why, where and when) [Hol93]. This model is based on answering three separate questions: why is the application performing poorly, where is the bottleneck and when does the problem occur. Such approach allows for quickly and precisely isolating a performance problem without having to examine a large amount of information.

To minimize the intrusion inserted into the application and perform more precise analysis, Paradyn supports the changes of the instrumentation during run-time. It provides a user with possibility to control data collection manually. However, the tool also contains a special module called Performance Consultant that liberates the user from making such decisions. Performance Consultant looks for performance problems, decides what data must be collected and when, and applies the instrumentation changes automatically during the application execution showing information to the user. The example of displayed results created during the searching phase is presented in Figure 3.5.

From the implementation point of view, Paradyn is divided into three parts: the Paradyn controller, the Paradyn daemons, and the application processes. The Paradyn controller performs searches for performance bottlenecks and supports the user interface to the rest of the system. The Paradyn daemon isolates all machine specific dependencies and serves as a stage between controller and application processes. Each application process is controlled by the Paradyn daemon and has the dynamically inserted instrumentation.



Initializing Search for Global Phase
CPUbound tested true for /Code,/Machine,/Memory,/Process,/SyncObject
topLevelHypothesis tested true for /Code,/Machine,/Memory,/Process,/SyncObject

CPUbound:/Code,/Machine,/Memory,/Process,/SyncObject

Fig. 3.5. Search history graph in Paradyn.

Although graphical representation is not a primary goal of Paradyn, it also contains a tool to display the results. Performance Visualizations can provide explanations of the program performance and since Paradyn is designed to work during run-time, visualized data is displayed “on the fly” as well.

3.4. Conclusions

Both approaches presented in this chapter provide automatic performance analysis of applications. The most important objective of these tools is to automate the performance analysis process and hence facilitate a developer the application performance improvement. Tools that support it use the collection of measurements gathered from the monitoring phase. Such tools are able to analyze collected performance data of the application behavior and explain to the user what happened during the application execution.

Static analysis provides post mortem analysis of the generated trace files, while dynamic analysis defines and processes the performance measurements and analysis at run-time. The second case is superior to the first one in that the trace files are no needed and the instrumentation can be added or removed automatically according to the actual program behavior. The instrumentation overhead can therefore be reduced and controlled. However, dynamic analysis as performed together with the application, might introduce more intrusion into the application execution.

Both kinds of tools present certain problems when the application behavior depends on the input data set or a studied application has dynamic behavior during execution. It can be noted that the recommendations and further code modifications for one run may not be useful for another.

Chapter 4

Dynamic performance tuning

This chapter introduces the background for automatic and dynamic performance tuning of parallel and distributed applications. We explain the fundamental concepts of dynamic optimization and introduce its requirements, constraints and applicability. Then, we compare the tuning approach to the automatic performance analysis. Next part determines our principal classification of the dynamic tuning approach. We describe the application knowledge problem while analyzing the performance and demonstrate our representation of the knowledge. We also extend dynamic tuning by the design of pattern-based framework that provides parallel application specification and can facilitate optimizations. We introduce the specific library called DynInst, that supports us with dynamic instrumentation. We describe the set of modifications that are possible to be applied dynamically. Finally, we describe available tools that were developed basing on the concept of computational steering loop and tools that support dynamic performance tuning of parallel programs.

4.1. Overview

Taking both kinds of performance analysis (classical and automatic) into consideration, we see the superiority and advantages of automatic analysis. However, none of these approaches exempts developer from the analysis of the output information nor intervention to a source code. Each solution that has been mentioned above requires developer to analyze generated results, combine them with application source code, change appropriate part of the source code, re-compile, re-link, and finally restart the program. The presented approaches requires a developer to have a high degree of the expert knowledge about the parallel application performance. Moreover, most of them needs a trace file either to visualize a program execution or to make an automatic analysis. When a post-mortem tuning is used it must be considered that the tuning for a particular run can be useless for another execution of the application. The application can depend on input data and for different data it can behave in different ways presenting different bottlenecks. Therefore, the results of analysis and offered optimization suggestions that are provided from one application execution might be inadequate for another one.

It can be concluded then that when there are dynamic conditions, such as variable behavior depending on the input data and/or variable behavior throughout the application execution, the tuning of distributed applications should be carried out dynamically. Moreover, the application can be executed on different hardware configurations. For example, a user can have a PC LINUX cluster with different numbers of PCs or can decide to add new PCs to the cluster, or change the old PCs for new more powerful ones, and so on. The application developer cannot guarantee that performance tuning for a particular system will provide the best possible performance when the system conditions are changed.

For all these reason, a new idea has arisen. The very convenient solution for developers would be to replace post-mortem analysis with automatic real-time optimization of a program performance. Instead of manual changes of a source code, it could be very profitable and beneficial to provide a developer with an automatic tuning of a parallel program during run-time. This approach would require neither a developer intervention nor even access to the source code of the application. The running parallel application would be automatically monitored, analyzed and tuned on the fly without need to re-compile, re-link and restart. The current application behavior would be considered and analyzed finding appropriate bottlenecks and possible optimizations. Therefore, if application had different behavior during different execution, dynamic tuning would adapt it taking into account changes of the behavior in current execution. Running applications under control of the dynamic tuning system would also allow the adaptation of their behavior to the changing environment and hardware conditions.

Figure 4.1 presents the model of the dynamic tuning approach. All the optimization phases are done during the application execution. The performance bottleneck search is not based on trace files, as the dynamic monitoring collecting necessary measurements provides them directly for the analysis process. While performing tuning, there is no need for manual application source code changes, because a tool that supports this kind of approach manipulates the application execution on the fly.

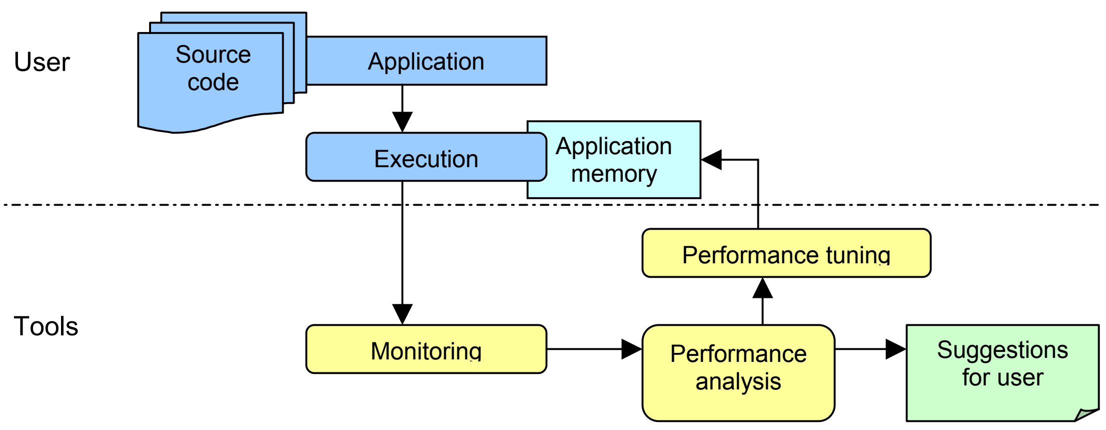


Fig. 4.1. The general view of the dynamic performance tuning approach.

Investigating dynamic tuning approach we have determined that dynamic and automatic optimization system should be based on the steering loop as the application behavior must be modified at run-time. Steering loop was defined as the capacity to control the execution of long-running, resource-intensive programs [Gu94]. Tools that support this approach allow users to study the behavior of the running application and manually change key application variables on the fly. We present some tools that are based on the computational steering loop later in this chapter.

Dynamic tuning system should provide the following services that cooperate among themselves during runtime:

- dynamic monitoring of the execution of a parallel program. This service provides the measurements collected from the application execution. It can be based on any of the performance monitoring technique: timing, profiling, event tracing. However, because of the goal to reduce the user intervention, instrumentation should be done automatically by the system. This service will relieve a developer of the manual code instrumentation and exempts from the invoking all these phases that must be carried out when the application source code has been changed (recompilation, re-linking and re-running). The measurement records are passed directly for the analysis.
- automatic performance analysis "on the fly". This service analyzes coming measurements, finds bottlenecks and gives solutions on how to overcome them. This service will replace the classical post-mortem analysis. To find problems and determine how to improve the performance, the analysis should have built-in performance knowledge about bottlenecks that are representative for the parallel applications. To be useful it should also include provision of detected problems and suggestions for a user.

- automatic program tuning during run-time. This service utilizes solutions given by the analysis process and automatically modifies a parallel application during the execution. It does need neither access a source code nor program recompilation, re-linking, restarting. Dynamic tuning will relieve the developer of the source code modification duties since the application execution is modified automatically “on the fly”.

4.2. Requirements and constraints

Investigating the performance tuning approach we have encountered many issues that must be taken into consideration. In this section we present the requirements and constraints that we must face up.

4.2.1. Parallel application control

In general, the parallel application environment is usually executed in a distributed environment that includes several computers. A parallel application consists of several intercommunicating tasks that solve a common problem. Tasks are mapped on a set of computers and hence each task may be physically executed on a different machine. Therefore, we must be able to control and optimize all the tasks on all the machines. To achieve this goal, dynamic tuning services must be distributed and executed on all the machines where the application tasks are running. In particular parts responsible for the monitoring and tuning are distributed since they work directly with the application tasks. Only in this way we will not loose any task and we will be able to provide the information about the entire application execution as well as tune any required task.

4.2.2. Global analysis

The parallel application distribution also means that it is not enough to improve tasks separately without considering the global application view. To improve the performance of entire application, we need to analyze globally the application performance. It implies the access to the information about all tasks on all machines. We must collect all the information extracted from the application execution at a central location. The analysis performed at this location considers all tasks and hence global application improvement is supported. The global analysis however, may be time-consuming due to the information collection and the performance analysis of this information searching for bottlenecks. Moreover, the application execution time can significantly increase especially if both – the analysis and the application - are running on the same machine. Many precaution then must be taken while developing the global analysis.

The global analysis approach is feasible for environments with a relatively small number of nodes and serves for the inter-node bottlenecks. If we consider performance problems related only to a given node without taking into account other nodes, there is no need for the global analysis. Additionally, to minimize the intrusion of the application execution, the analysis process should be executed on a dedicated and distinct machine. However, the dedicated machine processing the analysis becomes a bottleneck if the number of nodes gets higher. Both problems, the scalability and the local bottleneck can be solved by distributing the analysis process. For example, local analysis could be performed individually by all the nodes considering only the locally available information, while global analysis could resolve problems caused by inter-node relationships. In the scope of this work we consider only the global analysis. We do not present local analysis solutions and examples, but it can be a good extension for the future work.

4.2.3. Application knowledge

The fact that dynamic performance tuning is carried out at run-time implies one basic and specific constraint to be considered: the analysis and the modifications must be kept simple. A programmer can develop any application that might present variety of bottlenecks and hence the analysis and the modifications might be extremely difficult. Decisions have to be taken in a short time in order to be effective in the execution of the program. The changes cannot affect the correct functioning of the application or crash it. Therefore, the performed modifications must be carried out carefully to ensure that the application correctly continues its execution. The modifications must not involve a high degree of complexity because obviously, it cannot be assumed that any changes on any application in any environment can be performed without taking any precautions. All these factors limit the application of dynamic tuning.

For all these reasons, evaluation and modifications cannot be very complex. Since all the tuning must be done in execution time, it is very difficult to carry this out without previous knowledge of the structure and functionality of the application. As the programmer can develop any kind of program, the potential bottlenecks search can therefore be complicated. If knowledge of the application is not available, the applicability and effectiveness of this approach might be significantly reduced. To avoid many limitations it would be profitable to provide specific information about the application and how to detect and overcome its bottlenecks. It implies the description of what should be measured, how to analyze the application behavior and what to change. Therefore, we must take into consideration the problem of the application knowledge definition and provision. We also must determine what is possible to change in an unknown application.

4.2.4. Run time monitoring and optimization

As we have already mentioned, all phases of improving the application performance must be done "on the fly". The important issue here is to determine how to insert instrumentation and apply changes to the running program without accessing the source code. To be able to dynamically instrument the application, the code insertion must be defer till the application is launched. Runtime code modification cannot require the source code recompilation nor restart and hence it must be performed directly on the memory of the running application. To fulfill this requirements a special novel technique called dynamic instrumentation should be used. This technique permits the insertions of a piece of code into a running program and changes of current behavior. The advantage of the dynamic approach is that the instrumentation can be added or removed automatically according to the actual program behavior. The instrumentation overhead can therefore be reduced and controlled. For example, the on-line analysis can focus on the specific execution aspect and start with an initial instrumentation. Next, when some thresholds are exceeded, an additional instrumentation can be introduced to obtain more detailed information. Finally, when the problem is solved, the required measurements can be reduced.

This approach is implemented by a library called DynInst. We analyzed this library and we saw that it is possible to manipulate a running application and manage the instrumentation insertion/deletion. DynInst is appropriate for two purposes: first to dynamically monitor an application, second to apply modifications to optimize performance during run time. We present this library in Section 4.8.

4.2.5. Low intrusion

The intrusion must be minimized. Besides classical instrumentation intrusion, in "dynamic performance tuning" there are certain additional overheads due to monitoring communication, performance analysis and program modifications as all these tasks are performed in parallel to executing application. Dynamic tuning system should minimize the overhead it implies itself since it controls and changes the application. In particular the performance analysis should not be computationally intensive. The instrumentation used for monitoring should minimize or gracefully handle large volume of information. The dynamic instrumentation approach allows management of the instrumentation amount inserted into the application. Therefore, dynamic tuning system must provide this kind of instrumentation management.

4.2.6. Target bottlenecks

The process of measurement, measurement refinement, analysis and actuation takes itself certain amount of time. It may happen that when a solution is available, the bottleneck has already finished. Therefore, this approach is feasible for problems that appear many times during the execution. The system, although it misses the first occurrences, is able to adapt and prepare the application for the next time it enters a problematic code region. This fact may appear as a very hard constraint, because the bottlenecks that appear only once are not solved. However, when thinking about applications running for several hours there are certain code regions that are executed many times (iterations). Consequently, the main performance bottlenecks are those that appear many times during the execution.

4.3. Automatic analysis vs. dynamic tuning

On one hand, the two described possibilities – automatic analysis (static or dynamic) and dynamic tuning – can appear as opposite approaches. They provide a user with different possibilities and results. They cover different application ranges. However, from the other point of view, the last reason causes that they can be considered as complementary. Together can provide analysis of wider range of applications. There are also several methodologies that are common for both approaches, especially concerning performance model used for analysis purposes.

The static approach has the advantage when the applications have a regular and stable behavior. They can be tuned and once the tuning process has been completed, the application can be executed as many times as necessary without introducing any intrusion during the application execution.

However, there are many applications that do not have such a stable behavior and change from run to run according to the input data or even change their behavior during one single run due to the data evolution. If application had different behavior during different execution depending for instance on input data, the hints provided by a tool that supports post-mortem analysis could be improper for another execution. In this situation, the dynamic approach allows following the application behavior on the fly. However, similarly to static analysis, the dynamic analysis is based on a single run of the application. When the application behavior depends on the input data or on the iteration of the execution, the suggested recommendations can be inadequate for a further run of the application. Therefore, in such a case the most appropriate approach is dynamic optimization of the application as it tunes the particular run. These characteristics make the dynamic tuning approach relevant to grid systems, where applications are executed in highly dynamic environments.

On the other hand however, dynamic analysis and tuning requires a continuous intrusion into the program that is not necessary when applying post-mortem analysis to the stable behavior application. Moreover, if the analysis is carried out on the fly during the execution of the application, the information available and the time spent on the analysis is considerably restricted. It is caused by the need to inform the user about the bottlenecks quickly and modify the application efficiently in this particular run.

Next issue is related to the monitoring phase. Monitoring tools that generate trace files can create enormous files due to the execution of a real application that takes some hours or days. Therefore, files can be difficult to manage and analyze post-mortem in order to find the performance bottlenecks. Moreover, the use of a single trace file to tune the application is not completely significant, especially 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 another run of the application. This problem could be solved by selecting a representative set of runs, but provided suggestions would not be specific for a particular run then. In this case, we see the advantage of dynamic tuning approach, which does not need trace files when analyzing the application behavior.

Dynamic tuning introduces much more intrusion into the application, what is reduced in the case of post-mortem and even dynamic analysis. However, automatic analysis (static as well as dynamic) providing only recommendations requires the user to participate in the tuning phase manually and to have more experience and knowledge developing parallel applications. In opposite the dynamic tuning approach reduces significantly the participation of the user improving the performance on the fly. The user can then be exempted from the hard and complex tuning task.

For all these reasons, we can see that each approach has its advantages and disadvantages depending on the features of the application, generated results and user obligations.

4.4. Classification of dynamic performance tuning

The most useful dynamic tuning is the one that can be used to successfully optimize the broad range of different applications. It would be desirable to be able to tune any application even though its source code and application-specific knowledge is not available. However due to incomplete application information this kind of tuning is very challenging and at the same time the most limited.

The key question is ***what can be really tuned in an "unknown" application?***

This section presents our classification of dynamic performance tuning. We show tuning layers which can be optimized in the application. We also classify dynamic tuning approaches.

4.4.1. Tuning layers

The answer to the key question can be found by investigating how an application is built. In general, each application consists of the following layers as it is shown in Figure 4.2:

- **Application-specific code**
- **Standard and custom libraries (API + code)**
- **Operating system libraries (API + code)**
- **Hardware**

An application is based on a set of services provided by the operating system. The operating system is responsible for managing the hardware (CPU, memory, I/O devices and network) and software components of a computer system. These components constitute the resources of the system. Operating system provides a set of libraries so that the users of the system see it as a functional unit without having to be concerned with the low-level hardware details. The application uses the system calls (OS API) to request the operating system (kernel) to do a hardware/system-specific or privileged operation. For example, the UNIX system provides a large number of functions (about 60) that address broad range of basic functionalities such as I/O, file handling, memory management, process management, inter-process communication (IPC), time functions, and so on.

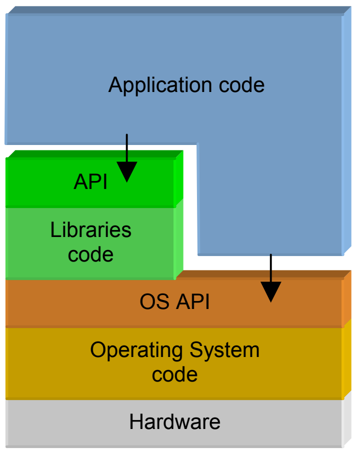


Fig. 4.2. Layers the application is built-on.

Besides that, the C/C++ applications use standard C/C++ libraries that support them with a variety of additional functions. This includes higher level I/O functions (i.e. buffered I/O), mathematical functions, string manipulation functions, time and date functions and other utilities. The implementation of some of these functions is based on OS services (i.e. system calls such as open (), read ()). Concerning C++ specific libraries, there is even wider set of provided functions that cover I/O streams, and standard class templates including vectors, queues, lists, strings, sets, maps.

Additionally, the application may use custom or third-party libraries that provide problem-specific functionality. They range from communication and message passing libraries (e.g. PVM [Gei94], MPI [MPI94]), database access libraries (OCI [L26]), numerical methods (ScaLAPACK [Bla97, L23], BLAS [Don88, L24], PETSc [Bal97, L25]) to programming frameworks (ACE [Sch94]). These libraries are developed to insulate the programmer from the low level details as they offer a higher level of abstraction and facilitate the design and development of high performance applications.

Finally, each application contains an application-specific implementation and consists of a number of modules that solve a particular problem in a given domain. An application supports different paradigms, its code uses variety of different data structures, functions, libraries, implements specific problem-domain algorithms. For example, to provide a matrix multiplication, a developer can build a parallel application on the top of PVM and C/C++ libraries and basing on the Master-Worker paradigm. Moreover, it can represent a data in a specific structure (e.g. as 2 dimensional vectors or lists) and an algorithm to data distribution and calculation must be used. For instance, having N workers, a total matrix is divided into N parts and distributed among the workers from the beginning or a matrix is divided into M parts and distribution is on demand when a worker finishes the partial calculation. The code might be written without functions, or encapsulated into variety of C function or C++ classes and methods.

To accomplish the performance expectations, a developer must tune the application choosing the best polices considering application requirements and the environment. Such tuning process requires a deep and detailed knowledge of the presented layers that is not necessary for developing applications. Moreover, these adaptations do not depend only on the application features, but also on the input data 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 application and its different layers on the fly during the execution.

We have distinguished 4 principal layers: application, libraries, operating system and hardware. All these different application layers may present different bottlenecks and hence may require dynamic tuning. The scope of this thesis does not include the hardware level, we consider only software optimizations.

Operating system performance issues commonly involve process management, memory management, file I/O and communication. We have identified two different methods that enable OS tuning in the context of a single process. The first method is the **adjustment of particular parameters of OS kernel implementation** to application needs and environment conditions. However, from the user-application point of view (i.e. user mode) it is possible only when there are adequate OS system calls that allow for such adjustment. Lower-level kernel code dynamic modifications would require to use dynamic kernel instrumentation techniques (such as KernINST [Tam99]), but this is out of the scope of this work. A good example for OS kernel parameter adjustment is the bunch of TCP/IP socket options. These options allow a user-application to tune the TCP stack behavior to particular needs of the application by means of setsockopt() system call. For example the application may enable or disable the Nagle's algorithm (TCP_NODELAY option) that decides if small messages should be grouped together before sending. This is beneficial for WAN networks with high latency, but typically slows down LAN communication.

The second method focuses on **tuning the code that inefficiently uses the underlying libraries.** This may refer to both the application-specific code and standard/custom library code that inefficiently use the OS functions. For example reading a big file with very small I/O requests (i.e. using non-buffered `read()` call) is a well known performance bottleneck and it is considered a bad usage pattern. In this case, the dynamic tuning system could detect this pattern by calculating `read()` call statistics and could tune the application by changing the request size (if possible) or dynamically insert buffering code.

Investigating the case of standard C libraries, we see many points that may be a cause of a bottleneck and hence may be good for optimizations. We can see that depending on the application needs, the library can be used in a bad or inefficient way. A well known problem while using the standard C library is the memory management. A performance bottleneck may occur when in an application there is a tendency to create quite large numbers of small objects. The memory allocations are usually based on the C heap allocator (`malloc()/realloc()/free()`). The C heap allocator is focused on medium- to large-sized objects (hundreds to thousands of bytes) and not on small chunk allocations. The solution is to rely on custom small-objects allocators – specialized allocators that are more efficient for dealing with small memory blocks (tens to hundreds of bytes).

Concerning STL standard C++ library there is a number of tunable options. One of the most common programmer's mistakes that may significantly affect the performance is to use the default capacity of dynamic containers (e.g std::vector). Default settings may be not well adjusted for different applications and their needs.

The next level of the application development is a custom specific-problem library. Such a library offers a higher level of abstraction, but is developed in a general way to be useful for a wide range of applications. This work focuses on improving the use of the library, not on modifying the library source code. The code modifications imply the recompilation process and our approach considers only dynamic and automatic changes done on the fly. For example, concerning common communication libraries as PVM or MPI, there are various possibilities to tune their usage as invoking or not the synchronization phase, eliminating the redundant synchronization, grouping or not messages sent to the receptor. Moreover, a parameter in a library function call can be switched according to the variable environment conditions. As an example we can put here the PVM function `pvm_initsend()` which contains a parameter that sets the encoding mode. By default PVM encodes data using XDR standard, because it cannot know if there are heterogeneous machines. If the messages are exchanged between homogeneous machines, the encoding phase can be skipped what allows for avoiding the encoding costs. Other possibilities to tune a custom library are selection of the most adequate policy or adjustment of a policy. For instance Adaptive Communication Environment (ACE [Sch94]) implements a set of design patterns that simplify the development of communication software. It provides C++ wrappers, frameworks or classes categories that perform common communication tasks as event demultiplexing, connection establishment, dynamic configuration of application services. ACE comes with a set of configurable services, but typically they are selected statically at startup, for example, a number of threads per request or a number of threads in pool.

Till now we can see that in these layers the optimization process may be based on the well known features characteristic for the operating system and libraries. Investigating operating systems and libraries it is possible to find their potential drawbacks. Some of the problems found can be tuned simply adjusting specific parameters (in case a function that adjusts them is available), other by tuning the bad and inefficient usage. All these tuning options are possible, because we can take advantage of operating system features and library implementation knowledge. We can focus on a set of problems related to paradigms used to implement the application that are common to many applications. For each drawback then the set of specific information can be determined to improve the application performance, such as measure points, performance model and tuning points/actions/synchronization. Problem that occurs due to the drawback of operating system's or libraries' layers can be eliminated, since such a problem is well known for any application that uses them. The developers can concentrate on designing and developing an application, and submit it with the input data. Once the application has been developed, the dynamic tuning can ensure a good performance. It takes care of controlling the application execution and optimizes different options according to the application and environment needs.

Tuning the application code is the most complex, due to the lack of problem-specific knowledge. Each application-specific implementation can be totally different and there might be no parts common for many applications event though they provide the same functionality. The application can be tuned using different techniques (parameter adjustment or algorithm selection) but only if there is a knowledge about its internal structure. Therefore, to optimize the application layer, dynamic tuning should be supported in some way with all necessary information about the application such as measure points, performance model and tuning points/actions/synchronization There are a number of application-independent code optimization techniques, but all of them operate at the lower level. The examples include dynamic function inlining (as it is provided by Java HotSpot [L27]), static code reordering (e.g. moving error handling code to the last else statement), static data rearrangement and so on. However, this kind of changes is out of the scope of our work.

One of the well known parallel application optimization technique is function call reordering. For example, in the PVM application function one of the problem may appear when one process blocks a message to be sent as it is waiting for a result from other process. Therefore, function `pvm_send()` could be invoked before `pvm_recv()` and hence the blocking would be eliminated. However, this change can be performed only if there is no data dependence. This phenomenon occurs when two memory accesses may refer to the same memory location and one of the references is a write. If these operations are forced to run in parallel, they may cause incorrect execution, as variables used for calculations may be utilized before they are updated. The example data dependence is shown in the following code:

```
for (i=1; i<=n; i++)
{
      a[i] = b[i] + 2;
      c[i] = a[i+1] + d[i];
}
```

Concerning the previous PVM example, `pvm_send()` can be invoked before `pvm_recv()` only if a content of a message to be about to sent does not depend on a result that will be received. If so, the function call cannot be reordered because it would cause an incorrect functioning. We see, that application tuning cannot affect the correct functioning of the application or crash it. Modifications performed on the fly must be carried out carefully to ensure that the application correctly continues its execution.

Summarizing, we presented different layers on which an application is built. Looking at Figure 4.2 the upper the layer is, the more specific information about the application is required. The lower the layer is, the more generic information is available. If we consider for example the operating system layer, it has many well-known information that can be used for any application. Such issues do not depend on the application structure, they are general. The library layer is already not so generic, but it also contains generic and common information that can be extracted without dependencies on the application. However, an application can be implemented in different ways, data dependence can occur and hence it is obvious that it does not have common and generic solutions.

4.4.2. Tuning approaches

Additionally to the answer to the key tuning question given by the tuning layers, the complementary answer can be found by investigating how an application is given. The classification we present in this section considers the available application knowledge. In our work we defined 2 principal approaches:

- **Automatic**
- **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 the tuning (no changes are introduced into the source code). In this approach, the key question is what can be changed in an unknown application. The automatic approach is more suitable for the tuning layers such as operating system and libraries. We can find there many general tuning options common to many applications. We can focus then on a set of problems related to paradigm used to implement the application as well as low-level functionality. For each particular problem, all the necessary information such as what should be measured, in what manner analyzed, what and when should be changed can be provided automatically.

In the cooperative approach, we assume that an application is tunable and adaptable. This means that developers must prepare the application for the possible changes. A programmer could prepare the application for tuning, for example by providing different implementations of certain algorithms and letting the tuning system select the most suitable one for the existing conditions. 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.

Another alternative is the application framework that hides all the communication mechanisms and provides tuning-aware implementation [Ces02]. This alternative allows the programmer to concentrate on codifying the application-related issues, hiding the low-level details. Moreover, it facilitates the dynamic performance tuning, providing all necessary information about the developed application. We present this idea in the Section 4.6. The cooperative approach is suitable for the application tuning layer as this is the less generic layers and many information should be known about the internal construction.

4.4.3. Alternative tuning approaches

As we have seen, our approach considers 2 different application treatment: as black box or as adaptive to be tuned. Both take use of the available knowledge (provided automatically or by the developer) and use a performance model-based analysis. Another approach to provide a dynamic tuning is dynamic feedback [Din97]. This is a technique that produces different versions of the same source code and each version uses a different optimization policy. The generated code alternately performs sampling phases and production phases. Each sampling phase measures the overhead of each version in the current environment. Each production phase uses the version with the least overhead in the previous sampling phase. The computation periodically resamples to adjust dynamically to changes in the environment. Dynamic feedback is used in Active Harmony [Tap02] described later in this chapter. This project bases on integration of different libraries with the same functionality. To finding the best solution, Active Harmony uses a heuristic algorithm. A similar solution is applied in RAS (Runtime Algorithm Selection) [Bor03] that provides an algorithm selection web service and meta-scheduler services during runtime.

4.5. Application analysis based on knowledge

Once we have classified dynamic tuning and presented examples of optimizations, we can focus on the problem of how to improve the performance analyzing applications. The purpose of the analysis is to examine application behavior basing on the collected measurements, identify performance bottlenecks, and give concrete solutions that overcome these problems. The analysis requires many information that allows for the application behavior determination and detection of the performance problems among coming measurements.

The application behavior can be characterized by an analytical performance model. A performance model helps to determine a minimal execution time of the entire application as it allows for prediction of the performance of that application. There are other possibilities to analyze the application behavior and improve its performance, as for example heuristics approaches. In such solution, some special parameters may be controlled, determined automatically by searching the parameter value space using heuristic algorithm and finally changed during run time. Heuristic algorithms neither determine nor predict the optimal application behavior. The goal while using them is to test the application behavior and find the best one changing the value of parameters. Our work concentrates on the analysis based on the performance model and rules.

Such a model can contain formulas and/or conditions that facilitate the calculations and determination of the optimal behavior. These formulas need as an input a set of information – measurements extracted from the application execution. Basing on the measurements and applying adequate formula, the performance model can provide the optimal behavior of the application, for example the optimal value of some parameter. Finally, the application can be tuned changing an appropriate parameter and its performance is supposed to be improved immediately.

For example, in the Master/Worker paradigm, the well known bottleneck is the number of workers involved in the work processing. Determining the performance model of such an application, it is possible to find an adequate number of workers. The performance model can require to measure the time taken by each worker when processing the data and the time when the master is waiting for results. Analyzing the measurements, the idle time of the master process can be detected. This situation means that the master process is mostly waiting for the results. Applying the formula to calculate the number of workers, the analysis can give as a result the optimal number. If this number is to increase, the application parameter that represents the number of workers must be tuned.

As we have described in Section 4.4.2, we distinguish two tuning approaches: automatic and cooperative. On the one hand, we consider as an effective solution automatic extraction of as much well known information as possible from the unknown application. It can be done only if there are well known problems and the performance model can be automatically determined. On the other hand, the provision of the application-specific knowledge is done cooperatively with a user. In this case, the developer determines a performance model of the application.

To make these two approaches homogeneous and to provide possible and effective optimization on the fly we decided that the application should be represented by a set of clearly defined information required for the monitoring, analysis and tuning. We assume the following principal terms and definitions:

- **measure points** – they determine what must be monitored in the application; a point tells where the instrumentation must be inserted to provide measurements.
- **performance model** – it helps to determine an optimal execution time of the entire application. It 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, tuning actions, synchronization** – they determine what and when can be changed in the application to obtain its better performance; 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. The synchronization specifies how and when the tuning action must be invoked to ensure the correctness of an application.

Figure 4.3. shows the concept of the dynamic tuning supported by the application knowledge in the form of measure points, performance model and tuning points/actions/synchronization.

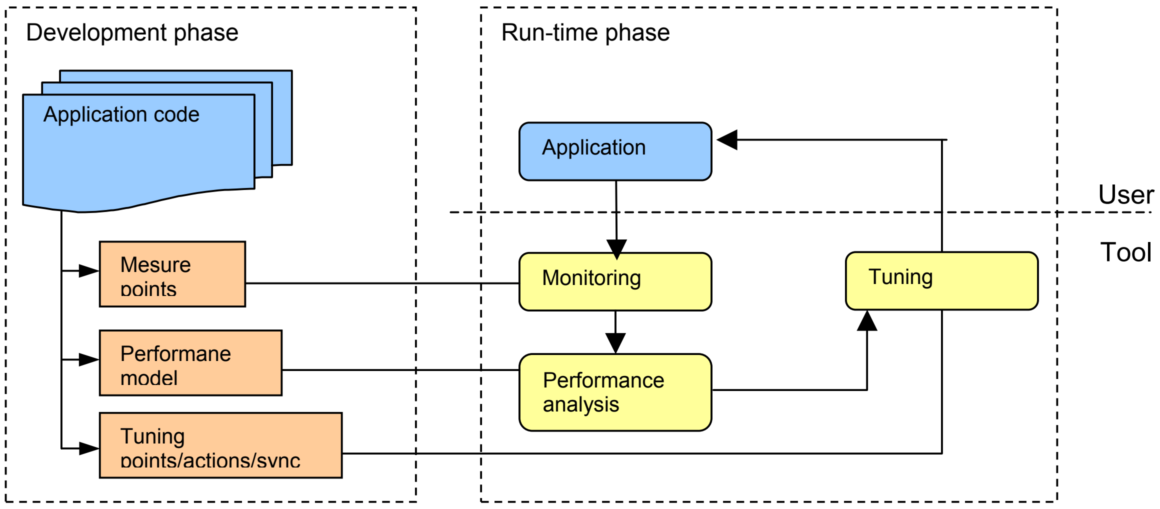


Fig. 4.3. Dynamic performance tuning and application knowledge.

4.6. Dynamic tuning supported by application specification

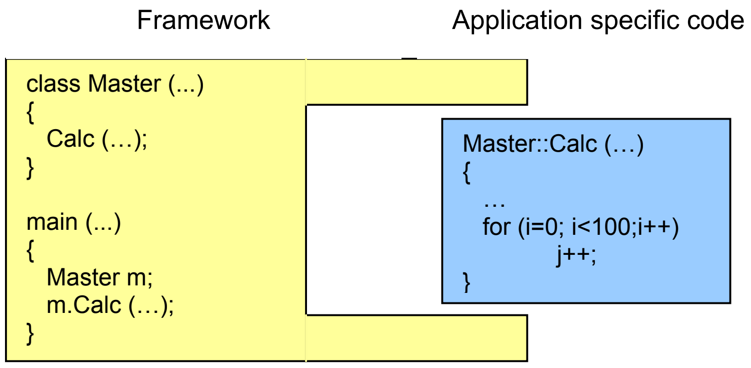
As we have mentioned before, dynamic and automatic tuning implies the analysis and optimization to be simple and effective. It is very hard to do it without a previous knowledge about the structure and functionality of the application. A good solution would be to know the application specification. Therefore, we propose an application development framework based on parallel patterns that provides our dynamic tuning with information about the internal structures of application [Ces02]. Moreover, the framework facilitates the programmers in design and development phases of their application. The users are constrained to use a set of programming patterns, but by using them they skip the details related to the low level parallel programming. All low level details of the communication library are hidden to the programmer. In this sense, using our framework API the programmers just have to fill those methods that are related to the particular application being implemented. They must indicate the computation that each process has to perform, specify data that must be computed and determine communication relationships of the processes. The whole conceptual loop, namely application design tool together with the dynamic performance tuning, allows the programmer to concentrate on the application design without taking into account low level details of the implementation and not to worry about the program performance.

4.6.1. Definition of parallel patterns and framework

For the purposes of this document we assume the following terms [Gam95]:

- *Design pattern* – is a problem/solution pair in a context. It can address design problems at many levels (however usually at low-level design issues). Pattern emphasizes reuse of design rather than reuse of code. It captures the static and dynamic structures and collaborations of successful solutions that distinguish them from poor ones when building applications in particular domain. Typically pattern provides description in a common notation together with design and advice for developers for better implementation of an application that will contain this pattern.
- *Framework* – addresses overall program organization. It contains a set of components that collaborate to provide a reusable architecture for the family of related applications. It tends to be more detailed and domain-specific providing a range of particular low-level functions. It emphasizes reuse of code as well as design. Generally, framework is a semi-complete application.

The programmers use a framework in order to build a complete domain-specific application. They provide implementation of only some particular components (classes, methods, functions) that are required for specific application functionality. The conceptual model of the framework is shown in Figure 4.4.



```
class Master (...)
{
   Calc (…);
}

main (...)
{
   Master m;
   m.Calc (…);
}
```

```
Master::Calc (…)
{
   …
   for (i=0; i<100;i++)
            j++;
}
```

Fig.4.4. Conceptual model of a framework.

Our research area is parallel programming, so we are focused on parallel patterns. The most known parallel patterns are [Vli95, L28]:

- *Master-Worker* – the pattern describes this kind of algorithms that are formed of a master process and some number of identical workers. It is used to describe concurrent execution by a set of independent tasks. Parallel applications that implement this pattern are called *embarrassingly parallel* because once the tasks have been defined, the potential concurrency is obvious.

- *Pipeline* – the pattern is used for algorithms in which data flows through a sequence of tasks or stages.
- *SPMD* (single program, multiple data) – the pattern describes an algorithm where a number of tasks execute the same program in parallel, but each task operates on its "own" data.
- *Divide & Conquer* – this pattern is used for parallel applications based on the well-known divide-and-conquer strategy. Concurrency is obtained by solving concurrently the subproblems into which the strategy splits the problem.

4.6.2. Parallel pattern-based framework

Our proposed parallel pattern-based framework supports a user during the parallel application development phase. It provides an API that is based on the object oriented methodology and implements patterns encapsulating their behavior, and the communication details. When a developer builds the application in our environment, he/she can choose what kind of parallel pattern is to be implemented in the application. Apart from the general structure of the application, the developer must indicate the computation done by all application parts, the data structures to be processed, and communication relationships. To develop the complete application, programmers use the API of the framework and provide their own implementation of particular classes and/or methods. The framework builds the application adding low level details of the implementation that are generated automatically depending on the chosen pattern.

All these parallel patterns that are implemented by the framework, their structures, behavior and possible performance bottlenecks are well investigated. Therefore, we can take a use of the research results and determine very useful information for dynamic tuning purposes. For each kind of application created in our framework, we analyze its specific problems basing on well-known parallel pattern bottlenecks. We are able to determine these application parts that from the one hand, can cause problems and from the other hand, can be changed in order to improve the performance. The framework can provide then the dynamic tuning with measure points, performance model and tuning points/action/synchronization. Using this knowledge, the dynamic performance tuning is simplified, because the set of performance bottlenecks to be analyzed and tuned are well known and related to the parallel patterns offered to the user.

4.6.3. Conceptual architecture

Even though the dynamic performance tuning is supported by the application framework, its the conceptual loop does not change. There are still three parts responsible for program optimization: monitoring, analysis and tuning. However, these parts have now significant amount of information about the application provided by the framework. Figure 4.5 presents the interactions and data flow in dynamic tuning while supported by application specification framework. The main components are:

1. Application Framework – it provides an API that offers support to the user during the application development phase. For an application based on chosen pattern, the framework generates for dynamic tuning all important and necessary information: measure points, performance model and tuning points.
2. Monitoring – it inserts the instrumentation into specified by the framework measure points.
3. Analysis – this part analyses parallel application using received events and the knowledge given by the framework. It knows performance model of the analyzed application. Therefore, it can directly start the analysis focusing on well-known bottlenecks of a given model, instead of wasting time on performing the initial search of the problem space.

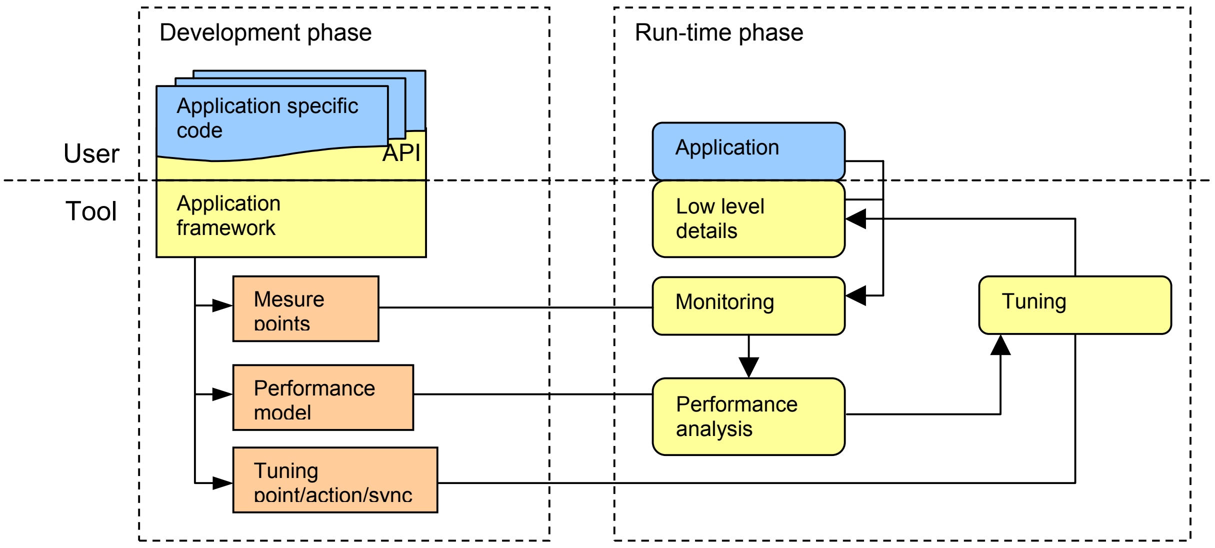


Fig. 4.5. Dynamic tuning environment supported by application specification framework.

4. Tuning – this part uses the solution given by the analysis and the tuning points and actions provided by the framework. Then it automatically manipulates the running application applying a given action on a given point.

4.7. Dynamic instrumentation

"The normal cycle of developing a program is to edit source code, compile it, and then execute the resulting binary. However, sometimes this cycle can be too restrictive. We may wish to change the program while it is executing, and not have to re-compile, re-link, or even re-execute the program to change the binary." [L10]

We have already described the general overview of the dynamic tuning approach as well as our principal definitions and classifications that we determined investigating this area. Now we will focus on a dynamic instrumentation technique that we decided to use to support our optimization approach. This section presents an overview of DynInst, an API for run time code generation. DynInst supports dynamic instrumentation and permits the insertion of code into a running program. We describe the API introducing its features, used abstractions and how to insert instrumentation into the application during run time. Then we show the small example of DynInst usage and we briefly describe the internal implementation issues of the library. Finally we present a commercial application of DynInst called DPCL developed by IBM.

4.7.1. DynInst overview

The principle of dynamic instrumentation is to defer program instrumentation until it is in execution and insert, alter and delete this instrumentation dynamically during program execution. This approach was firstly used in Paradyn tool that we have described in Chapter 3. In order to build an efficient automatic analysis tool, the Paradyn group developed a special API that supports dynamic instrumentation. The result of their work is called DynInst [Buc00, L30]. DynInst is an API (Application Program Interface) for runtime code patching. It provides a C++ class library for machine independent program instrumentation during application execution. DynInst API supports a programmer when building the application that will instrument another application during run time. The API is based on object-oriented technology and provides a set of classes and methods that allow a user to:

- attach to an already running process or starting a new process
- create a new piece of code
- access and use existing code and data structures
- insert created code into the running process
- remove inserted code from the running program

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. A very important issue is debug information. DynInst can manipulate a program during run time, but with one condition: it requires that the instrumented program contain debug information. The API needs symbolic debug information to be able to locate procedures and variables in the instrumented application. Therefore the instrumented program must be compiled with appropriate option to enable this information.

The goal of this API is to provide a machine independent interface. This allows the same instrumentation code to be used on different platforms. The newest version of DynInst 4.0 supports the following platforms: Sparc Solaris, x86 Solaris, x86 Windows NT, x86 Linux, Alpha (Tru64 UNIX), MIPS IRIX, Power/PowerPC (AIX).

4.7.2. Abstractions

The DynInst API is based on the following abstractions:

- **mutatee** or **application** – a program to be instrumented.
- **mutator** – a separate program that modifies an application process via DynInst.
- **point** – a location in a mutatee where a new code can be inserted, i.e. function entry, function exit, subroutine, long jump.
- **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 Abstract Syntax Tree (AST). It can include conditionals, function calls, loops, etc.
- **thread** – a thread of execution (it means process or a lightweight-thread).

- **image** – it refers to the static representation of a program on a disk. Each thread is associated with exactly one image.

Abstractions used by DynInst and their relationships to each other are presented in Figure 4.6.

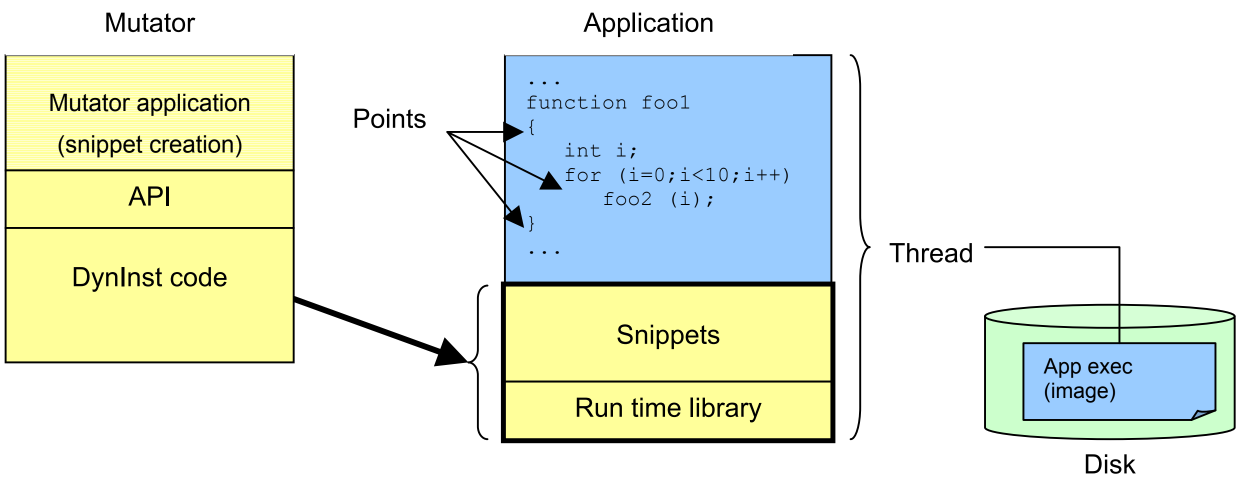


Fig. 4.6. Abstraction used in DynInst.

To be clear, we present definition of the Abstract Syntax Tree [L31]: the Abstract Syntax Tree (AST) is the hierarchical representation of the semantic features of a program based on its abstract syntax. The abstract syntax defines only what is done rather than how it is done. The goal of the abstract syntax tree is to store intermediate representation of input between multiple passes of the compiler. AST has the "same semantics as the source language they represent." This is an advantage over the byte-code so that it is possible to quickly establish the intention of the program at every level.

4.7.3. DynInst usage

To insert instrumentation dynamically, a user of the DynInst library must do the following steps in the development phase:

1. A mutatee executable file must be available. There is no need for the source code of the application, neither special compiler nor linker options set (only debug information is required).
2. A mutator must be implemented in a special way: using appropriate DynInst classes.
3. The mutator must implement snippets (instrumentation) using DynInst classes.
4. The mutator is compiled and linked with DynInst library (all these issues are described in the DynInst guide [Hol03]).

5. Then the mutator is started.

Then the following steps are performed automatically by DynInst library during run time:

6. The DynInst library is dynamically loaded to the mutator address space.
7. The mutator, via DynInst, creates an application process (or attaches to an already running one).
8. DynInst automatically attaches its run time library and snippet code to the address space of mutatee process.
9. At all specified points of the mutatee, DynInst inserts calls to the snippet code.
10. When the function has a snippet call inserted at the entry and is being executed, first a snippet code, then an original function code are performed.

Figure 4.7 presents all described before steps that are performed during a development phase and run time phase in order to insert instrumentation dynamically into the application via DynInst.

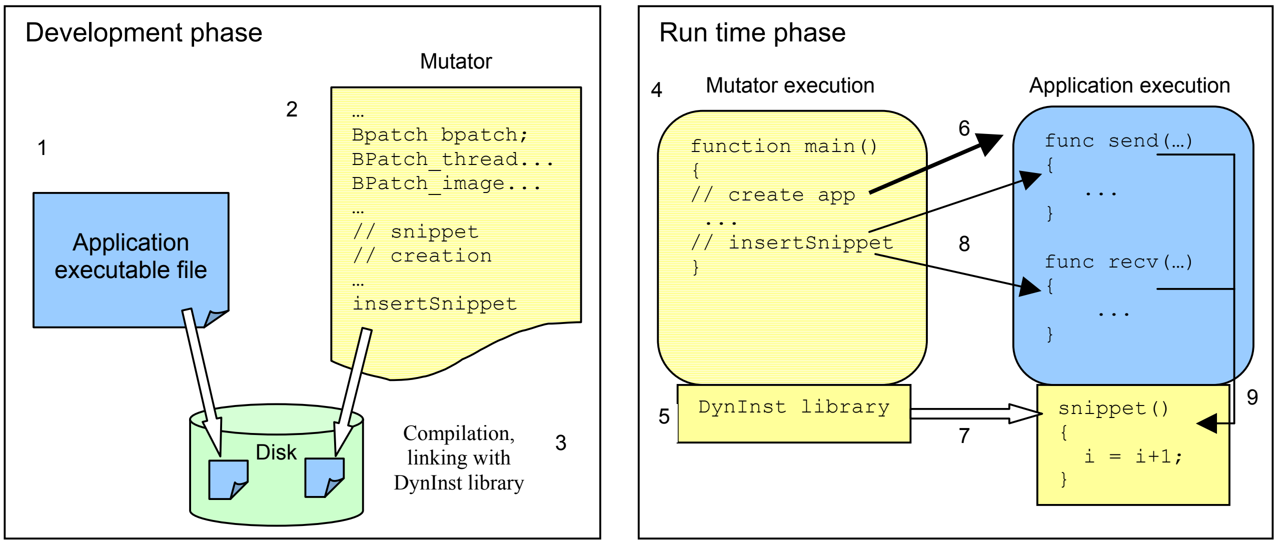


Fig. 4.7. Steps of the application instrumentation when using DynInst library.

Creating snippets is the most difficult part when using the DynInst library. Therefore, we have focused on the instrumentation creation. However, DynInst also allows a programmer to disable and remove the instrumentation. It is also possible to alter a semantic of a program changing a call to specified function to be a call to another one. In order to perform these operations, the developer utilizes appropriate corresponding methods of library classes. For example, if the instrumentation must be removed from the process, the developer invokes only one method and DynInst automatically during run time removes snippet calls from the specified points.

The API is organized as a collection of C++ classes. Each class provides a set of methods that can be invoked by the user of the library. In this document we do not present the classes and their methods. Listing of all classes, methods, parameters and descriptions is available in the “DynInst API Programmer’s Guide” [Hol03].

4.7.4. Example snippet creation

Here we present the creation of an example snippet. If we wished to record the number of times a given function was invoked during the application execution, we would define a point and a snippet in this way:

- point – first instruction in a given function (e.g. function: `pvm_send()`, location: entry)
- snippet – a statement to increment a counter

Generally, to increment a variable a programmer would generate (in C):

```
int var;
var = var + 1;
```

However, a snippet must be implement as AST. Figure 4.8 shows two views of AST: a conceptual and by means of DynInst classes.

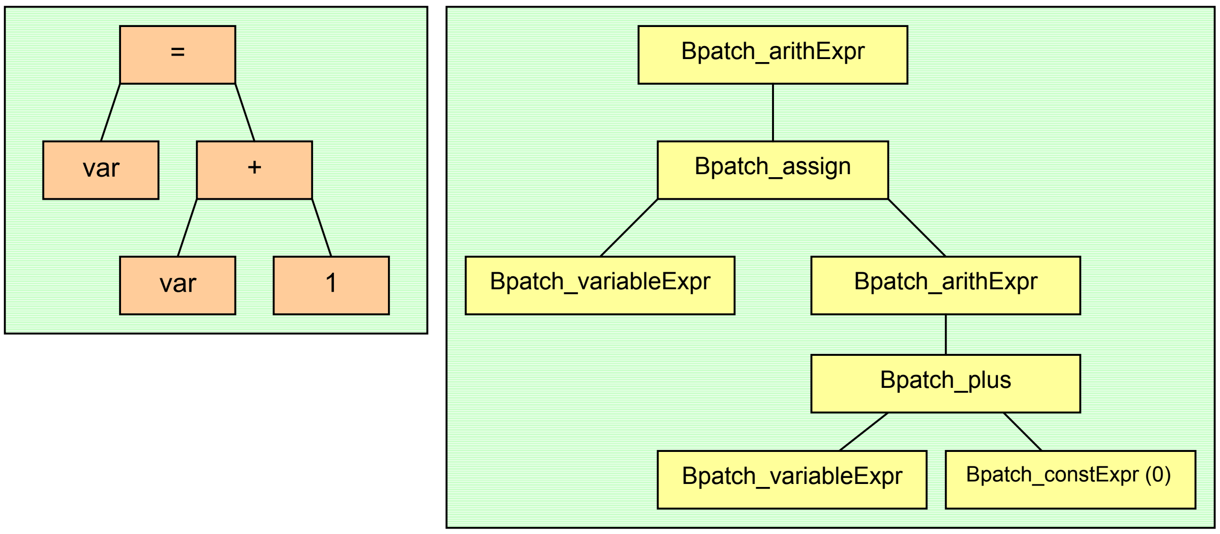


Fig. 4.8. Abstract Syntax Tree (conceptual and using DynInst classes) representing the arithmetical snippet that increments variable *var*.

Basing on the view of AST with DynInst classes, we present an implementation of the snippet that increments a counter. The code we write is:

```
BPatch_variableExpr counter = appThread->malloc ("int");

BPatch_arithExpr addOne (
        BPatch_assign,
        counter,
        BPatch_arithExpr (
                BPatch_plus,
                counter,
                BPatch_constExpr (1)
        )
);
appThread.insertSnippet (addOne, points);
```

First we allocate the memory in the mutatee address space for the integer variable (`BPatch_variableExpr counter`) which will be incremented. Then the object `addOne` of the class `BPatch_arithExpr` is created. This object represents an arithmetical expression and assigns (`BPatch_assign`) to the variable `counter` the result of another arithmetical expression (`BPatch_arithExpr`) that adds (`BPatch_plus`) to the variable `counter` a constant expression (`BPatch_constExpr`) with value 1. The last step inserts the defined snippet into thread at all specified points.

4.7.5. DynInst internal issues

In order to instrument an application during run time, DynInst library must perform special operations on the application executable file before the application start. Implementation of DynInst includes structural analysis of the binary searching for the possible points in the program and instrumentation management that allows code to be inserted and removed from the running program. In the following sections we briefly describe how DynInst performs these operations. The detailed description can be found in the technical documentation of DynInst [Hol97].

4.7.6. Structural analysis

Before a new process creation (or attaching to the already running one), DynInst library performs a structural analysis to identify instrumentation points in the application. DynInst extracts necessary information from the symbol table and by scanning the binary image. It generates a list of all possible points for each function where the instrumentation can be inserted. Each point is annotated with important information such as: point address, original instruction at the point. For each function DynInst defines weather the function represents a leaf or it creates a new stack frame.

An executable file is processed in several steps. First, DynInst maps the memory to the executable file and processes the symbol table to get the size and address of the code and data segments. It generates the following information: pointers to the code and data segments, list of symbols (functions and data objects) with name, type, starting address, size, etc. When the information about all functions is available, DynInst searches for instrumented points (entry, exit, call sites) for each function. The entry point of the function is the starting address obtained from the symbol table. The other instrumentation points are defined scanning the function code and searching for instruction that implements calls (call instruction) or exit (e.g. return).

4.7.7. Instrumentation management

In order to instrument a code during run time, DynInst generates instrumentation codes translating snippets into machine language codes. Then it places them into trampolines that reside in dynamically allocated areas in the application address space. Trampolines provide a way to invoke instead of the original code the newly generated code.

Dynamically allocated areas are provided by a dynamic linked library of DynInst. It contains utility functions and two large arrays loaded into the application address space. Both arrays are used for dynamically allocated regions of memory. One is used for instrumentation variables, and the other to hold instrumentation code (on many platforms instructions and data are kept in separate regions of memory).

Once the code is generated and placed in trampolines, DynInst must tie it with the application. Carefully modifying the code to branch into the newly generated code is the most difficult part of inserting instrumentation. To tie the generated code with the application, DynInst stops the application process and installs the code into the required point in the application address space. DynInst employs the same basic operating system services as used by debuggers (`proc filesystem`, `ptrace`). These services provide a way to control process execution, and to read and write the address space of the application. `proc` is a filesystem that provides access to the image of each active process in the system. `ptrace` allows a parent process to control the execution of a child process.

DynInst extracts appropriate instructions from the instrumented point of the application and relocates them into the reserved space. The original code is modified to jump to the base trampoline. The base trampoline contains the relocated original instructions, instructions to save and restore registers, slots where jumps to mini-trampolines can be inserted and jump to return to the application. The mini trampoline contains the instrumented code (snippet) and jump to return to the base trampoline. Figure 4.9 presents the structure of the base and mini trampolines and its relationship to the instrumentation point.

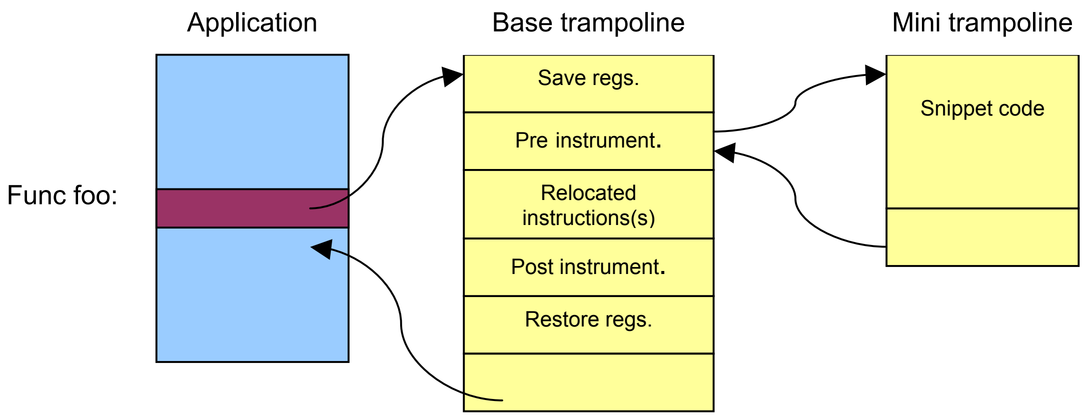


Fig. 4.9. Structure of the base and mini trampolines.

4.7.8. DPCL

DPCL stands for Dynamic Probe Class Library and it is the library that simplifies building tools for application performance analysis [Pas98]. It provides an infrastructure to reduce the cost of writing instrumentation. DPCL was developed by IBM Corporation [L6] in 1998. The library takes advantage of dynamic instrumentation provided by DynInst. DPCL is C++ class library build on the top of the DynInst. DPCL encapsulates DynInst functionality providing a programmer with possibility to use the higher-level abstractions comparing to DynInst ones.

Programmers build their end-user tools using DPCL library classes and methods. Then during run time they can establish connection with the application to be analyzed and manage the application instrumentation – insertion and deletion. DPCL is implemented as a distributed, asynchronous system that contains special daemons for providing services. When a tool based on DPCL is executing, it requests services from daemons via library calls (it calls methods of the classes from the DPCL library). The daemon translates those requests into actions and interfaces with the DynInst library to instrument and manage user processes. The instrumentation sends performance data back to the tool through the daemon. When the message is received it activates a callback function supplied by the tool for that purpose.

Instrumentation is defined by the user tool as a combination of probe expressions and probe modules. Probe expression is represented as an Abstract Syntax Tree, while probe modules are collections of functions written in a standard language (for example C) and compiled into object files, that are loaded into the application and called from a probe expression. The Figure 4.10 shows the conceptual tool architecture when using DPCL.

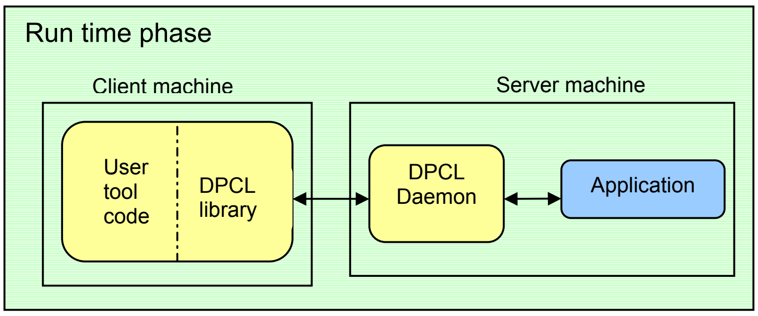


Fig. 4.10. A tool communication with a daemon through the DPCL library.

DPCL provides similar to DynInst services. We present only few of them:

- application and process management - create an application/process, connect to and disconnect from a running application/process (in this case application contains multiple processes); suspend, resume, terminate application; read/write application memory; open/close/read/write/seek application file
- instrumentation management – select/identify instrumentation; create/install/remove probe expression; activate/deactivate probe expression; periodically activate instrumentation
- communication between probes and tools in order to send data back to the client tool

All available classes and their methods are described in [Rob98]. DPCL library is working only on the IBM machines, namely RS/6000 Scalable POWERparallel Systems (SP – a scalable system arranged in various physical configurations, that provides a high-powered computing environment). As operating system, IBM uses its licensed version of the UNIX, namely AIX – Abbreviation for Advanced Interactive Executive [L32].

4.8. Dynamic modifications of an application

Once we know how we can change an application during run time, the next question is what we can change in a given application. As we have mentioned in Section 3.5 we define terms as tuning actions, tuning points and synchronization. Considering the possibilities of DynInst, we determine a set of tuning actions that can be applied on tuning points. A tuning point can be any point found by this library in the application executable, as function entry, function exit, call places. We consider the following tuning actions:

- **Function replacement** – in this method, all calls to a given function are replaced with a call to another function with an identical signature. The implementation of a new function can already exist in the application or operating system or it can be provided by our dynamic tuning in a dynamic library loaded to the process. The example tuning option is inside the memory management. When the standard C library function malloc() is not efficient in some circumstances, then all calls to this function can be replaced (if possible) with the calls to the custom function.
- **Function invocation** – an additional function call is inserted into the application at a specified point. From that moment on, each time this point is reached, the inserted function will be invoked. Function implementation is delivered as in the previous point. In general this kind of action is used for monitoring purposes, as the instrumentation must be inserted into the application to generate the information about application execution. As an example we can consider multithreaded application. To modify a shared variable, a thread must use a critical section (e.g. mutex) to synchronize access. This may become very costly if the operations of locking and unlocking are repeatedly executed inside a loop. In some circumstances (e.g. recursive mutex), it is beneficial to insert a mutex lock/unlock function calls before/after the loop. These additional calls amortize a cost of locking/unlocking inside the loop and reduce the synchronization overhead.
- **One-time function invocation** – a specified function is invoked just once. Function implementation is delivered as in the previous point. For example, if message buffering causes inefficiency, the Nagle's algorithm might be disabled through the TCP_NODELAY socket option. This action can be applied just once when establishing the TCP connection. Therefore, function call `setsockopt(socket, IPPROTO_TCP, TCP_NODELAY, optionValue, optionLength)` should be invoked just once to sets the option.

- **Function call elimination** – a specified function call is eliminated. It can be performed to remove unnecessary function calls. Obviously, it is required that such a removal does not affect the correct functioning of the program. This action can be used for example to eliminate redundant synchronization calls. Another example is removal of debug print() or flush() statements if their costs are considered to high.
- **Function parameter changes** – the value of an input parameter is modified before the function body is executed. As we have mentioned before, PVM uses for example special encoding while sending messages. The encoding is set by the PVM function call `pvm_initsend(encoding)`. If the encoding process can be avoided, this function should be always called with the appropriate parameter value `(pvm_initsend(PvmDataRaw)`).
- **Variable changes** – the value of a particular variable in the application is modified. The application should be aware that the variable is mutable. As an example we can put here the number of workers. The application must have special outside-known variable that represents a number of workers. If the variable has been changed, the application must be aware of that and apply the modification correctly.

The synchronization specifies when the tuning action can be invoked to ensure the correctness of an application. For example, to avoid reentrancy problems, race hazards or other unexpected behavior, a breakpoint can be inserted into the application at the specific location. When the execution reaches the breakpoint, the actual tuning action is performed. For example, the tuning action may include one-time function invocation `pvm_setopt(what, value)` that sets options of the PVM library. This function should be invoked before the message is sent. It cannot be called when the message is being sent, because it can cause reentrancy problems in PVM library implementation. Therefore, invocation must be synchronized with the application execution. The breakpoint can be inserted at the entry of function `pvm_send()`. When it activates, first `pvm_setopt()` call and then the actual `pvm_send()` call are performed.

4.9. Example tools

Dynamic optimization tools have the flexibility to adapt program execution to changing scenarios and differing hardware configurations. They provide the possibility to tune a program execution during run time, hence dynamic adaptation has been applied in many scientific domains.

First, several tools were developed basing on the concept of computational steering loop. Such tools allow users to study the behavior of the application under execution and manually change key application variables (e.g. resource allocations, program state, computational methods, data output). They have built-in the components for computation modeling, scientific simulations and visualizations. However, most of these tools are designed to allow the application semantics to be changed. They serve a user as a problem-solving environment (PSE), rather than performance tuning. Tools that are based on the concept of interactive computational steering are for example Falcon/MOSS, SCIRun [Par95], PPFS [Ree96].

In the dynamic tuning area, there are already a few projects that go toward automated performance optimizations and adapt applications to changing conditions automatically during run time. Based on measurements and analysis, the tool can improve the application performance during run time solving problems and adjusting them to better match resource requests. The most known tools are: Autopilot, Active Harmony and AppLeS. We present the most significant features of these tools indicating also their differences to our dynamic tuning approach.

4.9.1. Falcon / MOSS

Falcon was developed at Georgia Institute of Technology and it is a set of tools that collectively support on-line monitoring and steering of parallel and distributed applications [Gu95, L33]. It allows users to improve program performance by changing its attributes during run time, to experiment with different program configurations, to play "what if" games. It consisted of four major components: a monitoring specification mechanism, on-line information capture and analysis, program steering and graphical displays of monitoring information.

Using Falcon's monitoring specification language, programmers define specific sensors for capturing information. Users can express program attributes – ranging from single variables to application state – that will be monitored and on which steering will be performed. During execution Falcon permits users to capture specific information by the inserted sensors and analyze it. The Falcon project focuses on the monitoring with low latency and perturbation and hence the monitored information is performed before it is displayed to the user. The graphical user interface, the graphical displays and steering mechanism interact with the run time system to obtain processed monitoring information. Application can be steered by human users or algorithmically. Once steering decision is made, changes to the program attributes and state are performed by the steering mechanism.

The Falcon software was applied to scientific applications, especially in physics and atmospheric area. However, it is no longer an active research project. The group stopped the work under the FALCON project and started to create next-generation system MOSS that stands for Mirror Object Steering System [L34]. MOSS does not use traditional event flow and introduces a higher-level object-based abstraction into the monitoring and steering (Mirror Object Model).

4.9.2. SCIRun

SCIRun stands for Scientific Computing and Imaging [Par95, L35] and has been developed at University of Utah. This is a Problem Solving Environment (PSE), and a computational steering system in which large scale simulations can be processed. SCIRun allows a scientist or engineer to interactively steer a computation changing parameters, re-computing, and then re-visualizing. SCIRun allows computational steering to be applied to the broad range of advanced scientific computations, e.g. in medicine, physics.

SCIRun contains several built-in tools to close the loop and provide computational steering. First, this is a framework in which a simulation can be composed. A user can design and modify the simulation via a visual programming interface to a dataflow network. Then, such a simulation can be executed, controlled and tuned by interacting with the end user via a graphical user interface. Finally, SCIRun can display information using 3D graphics (see Figure 4.11).

Over past years, the group has developed two additional problem solving environments that extends SCIRun capabilities: BioPSE and Uintah [L36]. BioPSE adds modules and functionality for bioelectric field problems. Uintah targets large-scale simulations running on distributed memory supercomputers.

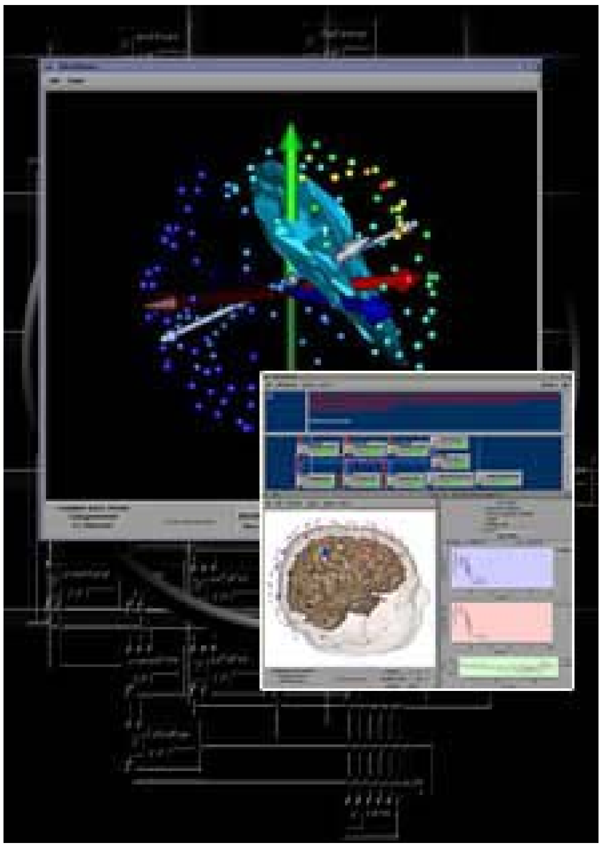


Fig. 4.11. Graphical user interface of the SCIRun environment.

4.9.3. Autopilot

The Autopilot [Rib98, L37] project developed at University of Illinois realizes adaptive control of parallel and distributed application. Autopilot bases on closed loop control and allows applications to be adapted in an automated way. It automatically chooses and configures resource management algorithms based on application request patterns and observed system performance. The Autopilot infrastructure is built on the experience and software from Pablo tool that was developed at the same university.

Autopilot provides a set of performance sensors, decision procedures and policy actuators. The toolkit uses distributed sensors to gather quantitative and qualitative performance data from executing applications. Every sensor has a set of properties defined when the sensor is created. These include name, type, identifier, network IP address and user-defined attribute-value pairs. Sensors can gather data using two methods. First, a sensor records data in response to procedure calls that have been inserted into the application manually by the programmer. Second, separate thread periodically awakes, reads application variables and returns to sleep. Sensors provide performance data for decision making and can transmit data using a variety of policies (e.g. transmit on demand, periodic update). The toolkit includes fuzzy logic engine that accepts performance sensor inputs and selects resource management policies based on observed application behaviour. Autopilot relies on fuzzy sets and use a set of IF-THEN production rules that map the sensor input values to the actuator output space. Finally, it realizes the results activating remote actuators. Actuators are remotely controlled functions that enable to invoke local functions or modify the values of application variables. Such actuator can change for example parameter values or resource management policies (e.g. file caching policy).

Moreover, Autopilot also provides mechanisms to manage local and remote tasks. The toolkit contains sensor / actuator manager and set of remote clients. Manager serves as a network distributed name server and supports registration by remote sensors and actuators. A client controls both sensors and actuators in associated tasks, receives data from sensors, and invokes actuators.

Autopilot contains a control interface to allow steering of infrastructure policies and application interactively or via automated decision procedures. The programmer can decide what sensors/actuators are necessary and then manually inserts them into the application source code. Autopilot contains a library of runtime components needed to build an adaptive application.

The approach applied in the Autopilot project is similar to our cooperative approach. However, it differs from the black-box approach where necessary measure and tuning points are decided and inserted dynamically and automatically by the tuning system. The Autopilot uses fuzzy logic to automate the decision-making process, while we decided to use simple, conventional rules and performance models. Moreover, in our case monitoring is based on the dynamic instrumentation where measure and tuning points are inserted on the fly. Using Autopilot a developer must prepare application inserting sensors and actuators manually into the source code.

4.9.4. Active Harmony

Active Harmony is an automated runtime tuning system [Tap02] that has been developed at University of Maryland. This 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 balancing during a single application execution based on the observed performance. The application must be Harmony-aware, that is, a programmer must apply changes in the source code and use the API provided by the system [L38].

Active Harmony focuses on the selection of the most appropriate algorithm. The system provides Library Specification Layer with uniform API. This layer integrates different libraries with the same or similar functionality. The user develops an application using this API, and hence the application contains 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 more efficient library and change tunable parameters to improve the application performance.

Active Harmony integrates two mechanisms that permit for automatic tuning. First, it exports a metric interface to applications, allowing them to access processor, network, and operating system parameters. This interface supports provision of data about the application performance and execution. Second, Active Harmony requires applications to export tuning options back to the system. A tuning option defines the expected utilization of one or more resources. The system can then use such a parameter to automatically optimize resource allocation basing on observed performance and changing conditions. Metrics and tuning options are specified as the part of Library Specification Layer.

The main part of the Harmony system is the Adaptation Controller. This component must gather information about the application execution, manage tuning options, propose the best changes, predict the effects of given modifications, and finally change appropriate parameters to improve the performance. 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. They base their minimization algorithm on the simplex method.

Active Harmony also includes graphic console that is shown in Figure 4.12. The console allows the user to manually optimize the application. The user can tune the values of the tuning parameters that are exported by this application. Moreover, user interface shows the performance function and the history of values.

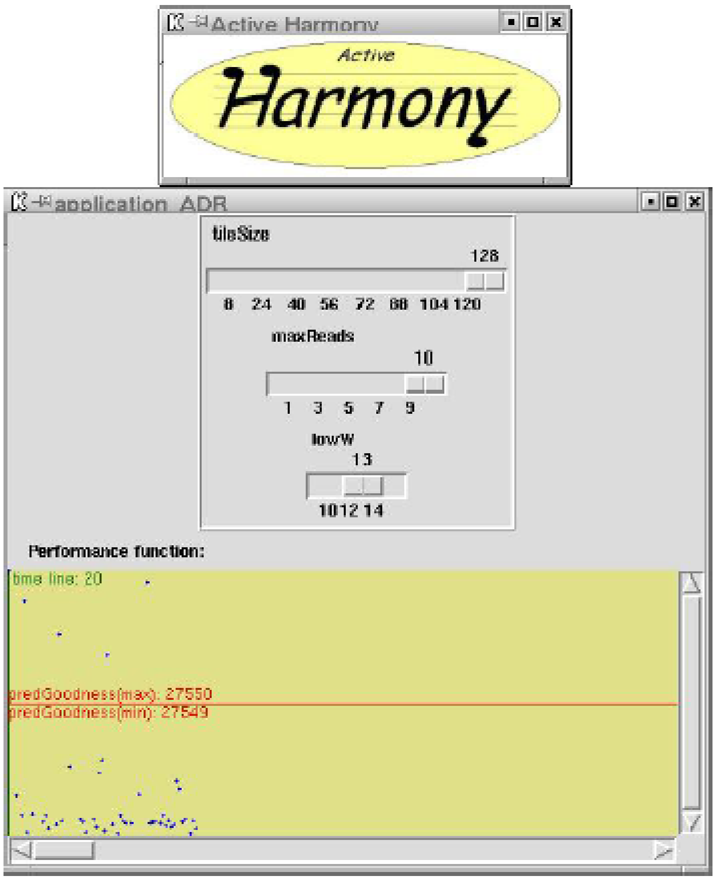


Fig. 4.12. Active Harmony user interface.

The Active Harmony system is conceptually similar to our cooperative approach. However, it differs from the automatic method that treats applications as black-boxes and does not require them to be prepared for tuning. Moreover, our dynamic tuning approach is based on the concept of measure points, performance model and tuning points. Such information can be defined by the user, but we also provides a dynamic tuning with certain predefined sets. Instrumentation is inserted into the application automatically during run time. Harmony bases on integration of different libraries with the same functionality. We use a distinct approach to finding the best solution. We do not use a heuristic algorithm, but performance models that provide conditions and formulas that describe the application behavior and allow the system to find the optimal values of the tunable parameters.

4.9.5. AppLeS

The AppLeS [Ber96] project from University of California 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. The programmer is supplied information via user interface 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.

AppLeS (Application Level Scheduler) is developed on agent-base methodology. Each application has its own AppLeS agent. Each agent contains static and dynamic information about the available resources and its function is to determine an application-specific schedule and implement that schedule on the distributed resources on metacomputers. An agents has built-in four subsystem and one single active agent called Coordinator. First, Coordinator takes resource information from the user via UI. Then, Resource Selector filters and chooses promising resources. Next, Planner generates schedule for a given resource configuration and Performance Estimator evaluates performance for candidate schedules. Finally, Actuator implements a chosen schedule on the target configuration in the resource management system. AppLeS is not a resource management system. It relies on the system as Globus or Legion and serves as middleware dynamically coordinating a customized schedule for the application.

Our approach is similar to AppLeS in that it is based on the automatic closed computational loop and it tries to maximize the performance of a single application. However, it focuses on the efficiency of resource utilization and performance bottlenecks that occur during the application execution rather than on resource scheduling.

4.9.6. Mojo, Dynamo

There are also dynamic optimization systems as: Dynamo [Bal00] – developed at Hewlett-Packard Laboratory and Mojo [Che00] – developed by Microsoft Research stuff. However, they approach to dynamic tuning differs from our one. These tools perform the run time optimization, but of a native instruction stream. The program binary is not instrumented and is left untouched during the system operation. This approach uses very low-level techniques of optimization.

4.9.7. HotSpot

The Java HotSpot [L27] is a product that has been developed by Sun Microsystem to provide the highest possible performance for Java applications. Traditionally, bytecodes are generated from Java programs and then interpreted during execution by Java Virtual Machine. To improve the program performance, Just-in-time (JIT) fast compilers are used that translate the Java bytecodes into native machine code on the fly. A JIT running on the end user's machine executes the bytecodes and compiles each method the first time it is executed.

The Java HotSpot VM provides adaptive optimization. It does not compile method by method, but it runs the program immediately traditionally using an interpreter. Then the HotSpot VM gather information about program hot spots (important optimization bottleneck) analyzing the code. Once it detects the critical hot spots in the program, they are compiled into native code and made inline (no function calls – code is inserted directly into the place where the function call is). The hot spot monitoring is continued dynamically during program execution. By avoiding compilation of infrequently executed code (most of the program), the Java HotSpot compiler can focus more on the performance-critical parts of the program, without necessarily increasing the overall compilation time. An example hot spot is frequency of virtual method invocation.

4.10. Conclusions

Parallel application tuning is very difficult and complex process if one wants to do it automatically and dynamically. There are many requirements that must be taken into consideration. They must be taken into account especially when developing efficient, useful and really helpful system. Moreover, it must be pointed out, that there is no possibility to apply dynamic tuning to any application in any environment. Solution based on the on the fly optimization generates several precautions and limits. The biggest effort must be put into the good definition of how the application can be tuned and what can be tuned there. We have seen the classification of dynamic tuning. We presented different layers on which an application is built and what is possibly to tune on each of them. It was clearly pointed out that the upper the layer, the more specific information about the application is required. We determined what exactly must be known about the application and how it can be treated, prepared for the optimizations and tuned.

Because of the complexity of the solution, there are not many tools that support application optimization during run time. Many of the existing tools go toward automated tuning, but require certain changes in the application. A real tuning tool should take into consideration the important issues as application analysis without knowledge about its internal structures and dynamic modifications of unknown application structures. However, such a solution is hard and complex. Investigating tuning area we have decided to develop our own dynamic tuning environment that supports cooperative approach of the application optimizations but it also goes toward the automatic tuning.

In this chapter we have also presented the novel, powerful technique called dynamic instrumentation provided by the DynInst library. We described the possibilities provided by the library proven with examples. The research on this technique and efforts put into the library development brought successful results. Dynamic instrumentation allows the flexibility in gathering data and offers the chance to significantly reduce measurement overhead. DynInst is a very efficient library available for many platforms and intrusion included into the running application is very small.

Chapter 5

MATE

This chapter presents an overview of MATE – Monitoring, Analysis and Tuning Environment. First we introduce motivation and goals of our environment. Next we present requirements that we have to take into consideration building dynamic tuning system. In continuation we describe issues of the MATE design and motivate decisions taken while designing. Finally, we present the system architecture and describe in details all system modules, their construction, functionalities and limitations.

5.1. Motivation and goals

We have seen that programmers of parallel applications must provide the best possible behavior of their applications if such an application is to fulfill a promise of the highest performance. Applications will be useless and inappropriate when their performance is under acceptable limit. However, programmers face up to many problems when improving a parallel application. Performance improvement is a complex and time-consuming task, and not feasible if it must be carried out manually by a developer. We have shown that one of the very promising approach is dynamic automatic tuning. It allows for the application performance improvement on the fly. Therefore, it is very beneficial to accomplish the performance expectations by using an automatic tuning environment.

Investigating the area we have seen that the proper solution would be to construct a tool that is able to automatically accelerate the application execution on the fly by adapting it to changing conditions. Such a tool would be really profitable especially when a parallel application is characterized by dynamic conditions, such as variable behavior depending on the input data and/or variable behavior throughout the application execution. A tool based on this approach would relieve developers from the complex manual tuning process.

In Chapter 4 we have indicated requirements for dynamic tuning in general, we have presented the classification, possible tuning techniques and examples. We can assume that all these techniques work also in practice and the application performance is improved.

However, it would be only theoretical dynamic tuning that would not present a real profitability. Without the practical probes we are not really sure if a tuning technique can be efficiently applied since we do not know for example, the intrusion of run time optimization or changing environment conditions (e.g. network load, machine load). Only experiments with existing software that permits dynamic tuning of applications will provide the total view of the dynamic tuning applicability. Therefore, many various practical experiments should be conducted on parallel applications to see if this approach really works, is effective, feasible, profitable, and can be used for a real improvement of the program performance. To perform practical experiments with different tuning techniques, we need a tool that will allow us to modify the parallel distributed programs during run-time.

We have defined detailed requirements and functionality that such a tool should provide. We have investigated existing tuning tools but we have missed there many important aspects. None of them treats applications as black-box because each tool requires an application to be prepared for tuning. There is no tool that would provide all the tuning phases simultaneously in a dynamic way. The performance analysis is usually based on the heuristic algorithms rather than on the concrete models that provide prediction of the application behavior. Therefore, we have decided to develop our own environment that will provide all required by us functionalities.

Our goal is not only investigation of dynamic tuning area for parallel distributed applications and presenting its applicability and effectiveness by means of only theoretical tuning techniques, but also the development of an environment that will support dynamic automatic tuning. Our goal is to help a user providing as much automation as possible reducing a degree of expert knowledge and user intervention. We want to prove that running distributed parallel applications under control of a dynamic tuning system would allow for the adaptation of their behavior to the existing conditions and for the improvement of their functionality. To support developers with dynamic performance tuning and prove its profitability we have created an environment that facilitates monitoring, performance analysis and optimization of parallel applications automatically during run time.

5.2. Overview of requirements

The goal of the tuning system is to improve the application performance and minimize the execution time by adapting the application to the available environment. To optimize the application during run time, to be efficient, useful and easy to extend, the tuning system must take into consideration many issues.

5.2.1. Target environment

Our investigation area is targeted to parallel and distributed scientific applications. Typically these applications are developed in PVM or MPI and they are executed in parallel environment that usually include several computers connected by network. A parallel application consists of several intercommunicating processes (a.k.a. tasks) that solve a common problem. These tasks are mapped on a set of computers and hence each task may be physically executed on a different machine.

Therefore, a tuning system that improves the overall performance of a parallel program must be able to control all its individual tasks on all machines. Moreover, as discussed in Chapter 4, it is not enough to optimize tasks separately but the global application view must be considered. This implies that a tuning system itself must be a distributed system. It must control individual processes of the tuned application and it must be able to gather global information about all associated tasks on all machines.

Frequently, the parallel environments used to execute the scientific applications are networked clusters of workstations or grid environments that connect various individual clusters and supercomputing centers. These environments are typically characterized by dynamic behavior (i.e. changing availability of resources, varying network load, etc.). The performance of applications even with static behavior may vary in these conditions. We think that it might be necessary to apply dynamic tuning in order to achieve satisfactory performance and hence we target our tuning system to this kind of environments.

5.2.2. Users

Parallel applications are able to provide high performance computing characteristics. They are used for solving many scientific problems such as the atomic interactions in a molecule, the simulation of the universe evolution or climate modeling. So biologists, chemists, physicists and many other researchers have become intensive users of parallel applications. Typically, these users are not experts in performance optimization. They would need a tool that facilitates them the performance tuning process. Therefore, our tuning system is targeted to the users of parallel applications, especially to the non experts.

As it has been indicated in Chapter 4, our work studies two basic dynamic tuning approaches: automatic and cooperative. In the automatic approach, the tuning system attempts to optimize an unknown parallel application and does not require its end-user to prepare it for the tuning. The programmer develops the application and then its user can execute it under control of the tuning system without any changes in the source code. In this case, the end-user is not required to provide any additional knowledge and he/she does not have to be a programmer.

In the cooperative approach, a parallel application must be tunable and adaptable. This means that developers must prepare the application for the possible changes, in some cases by modifying its source code. They must provide the system with the 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 in the application. Then the tuning system helps them to optimize their applications online. However, to provide the required knowledge a user must know the potential performance problem as well as the way of its detection and solution. Moreover, he/she must be able to implement an appropriate bit of code that allows the tuning system to tune the application. In this case the degree of user participation and expertise is much higher than in the black box approach and thus the group of the users of the tuning environment might be reduced.

5.2.3. Required system characteristics

The dynamic tuning system should have the following characteristics:

- **Online monitoring, analysis and tuning** – it is required that all phases of performance optimization are performed online, i.e. during application execution.
- **No source code, no recompile, no relink** – the source code of the application is not required for dynamic tuning. The application does not have to be recompiled nor linked with any additional libraries.
- **On-the-fly instrumentation** – the tuning system should be able to add/remove instrumentation code for monitoring and tuning during run-time.

- **Safe tuning –** the tuning methods should be kept simple. The changes cannot affect the correct functioning of the application. We cannot assume that an application can be modified without taking any precautions.
- **Black-box and cooperative tuning –** to experiment with both approaches, we require the system to support both methods. Concerning tuning layers that we presented in Chapter 4, we focus our work on the following layers: usage of operating system, usage of custom specific-problem library and application. The system must be able to adjust parameters, change algorithm and tune the code that inefficiently uses underlying libraries. Therefore, we require it to provide the set of tuning actions described in Chapter 4: function replacement, function invocation, one time function invocation, function call elimination, function parameter changes, variable value changes.
- **Low intrusion** – the goal of the tuning system is to improve the execution time of the program. Therefore its implementation should minimize the overhead it implies itself by controlling and changing the tuned application. The instrumentation used for monitoring should minimize or gracefully handle large volume of information.
- **Lightweight analysis –** the performance analysis process should be lightweight and not computationally intensive. It is recommended to keep the analysis simple to be able to take decisions in required time-frames.
- **Global application view** – the tuning techniques that affect the functioning of the parallel application may require the tuning system to base its decision on global knowledge of the tuned application.
- **Open and extendable –** the tuning system should be open and allow the developers to integrate new tuning techniques.
- **Easy to use** – from the user perspective, the tuning system must be easy to use. In the automatic (black-box) approach, the best option would be to simply execute the parallel program under control of the tuning system and let it do its work.
- **Portability** – due to variety of platforms used to execute the parallel applications, it should be possible to port the tuning system to different operating systems and support different communication libraries.

5.2.4. Assumptions and dependencies

As we have already mentioned, all phases of improving the application performance must be done on-the-fly. To fulfill this requirements we decided to use a novel technique called dynamic instrumentation. This technique permits the generation and the insertion of a piece of code into running program without accessing its source code. **We assumed to use the DynInst API** [L29], a library that provides platform-independent dynamic instrumentation. We have presented this library in Chapter 4. We use DynInst library for two purposes:

- dynamic monitoring – to provide the dynamic instrumentation phase. It is possible to manage (add and remove) a code (instrumentation) that collects information about the application behavior.
- dynamic tuning – to provide the dynamic modification phase. It is possible to change the code of the running application in order to improve its performance.

The DynInst library implementation is directed to applications written in C/C++ languages. Moreover there is a large number of scientific applications written in these languages that use PVM or MPI communication libraries. Therefore for the purpose of our work we assume to target our tuning system to **C/C++ parallel applications based on PVM communication library.** However, we require that the design of the system is open and could be extended to support applications that use other message passing communication libraries (for example MPI).

Finally we decided to implement the **tuning system on the UNIX platform**, because it is the de facto standard platform for scientific parallel application. However, the implementation should minimize platform specific dependencies to enable its port to different operating systems.

5.3. Functional requirements

Our environment is required to perform dynamic tuning. From the functional point of view we can distinguish three basic and continuous phases: monitoring, performance analysis and optimizations. All of these phases must be performed continuously, dynamically and automatically while the program is running. The environment should be based on the computational steering loop concept and should exempt a developer from intervention into the tuning process. Our tuning system should dynamically and automatically instrument and monitor a running application to gather information about the application behavior. The analysis phase should search for performance inefficiencies, detect their causes, give solutions on how to overcome them. Finally, the optimization (tuning) phase should dynamically modify the application by applying given solution. Moreover, during application execution, the environment cannot require access to the application source code. The running parallel application should be automatically monitored, analyzed and tuned without the need to re-compile, re-link and restart.

An issue of particular importance is the representation of knowledge that can be used to drive the dynamic tuning of the parallel application. This knowledge should be specified independently from the environment implementation, in order to enable the extensibility and the inclusion of new performance problems and their tuning techniques. We require the environment to provide a solution to this problem and specify the representation form for measure points, performance model and tuning points/actions/synchronization (see Chapter 4 for definition of these terms).

In the following sections we summarize the functional requirements of our environment:

- **Control the execution of the parallel application**
    - Start the parallel application
    - Attach to the running application
    - Control startup and exit of individual application processes
- **Automatically control application performance monitoring**
    - Decide what monitoring data is necessary
    - Decide where and how the required information should be collected
    - Request insertion/removal of instrumentation code
- **Perform application monitoring online**
    - Generate new piece of instrumentation code
    - Insert instrumentation code to the individual process at a specified location
    - Remove previously inserted code
- **Collect monitored data for analysis online**
    - Gather data generated by inserted instrumentation code from the individual application processes
    - Deliver performance monitoring data for analysis
- **Analyze application performance online**

  - Use externally provided knowledge (measure points, performance model, tuning points/actions/synchronization) to drive the analysis
  - Evaluate performance using collected monitoring data
  - Find performance problems
  - Decide if tuning is necessary
- **Control application performance tuning online**
  - Find out what should be changed in the application
  - Decide where, how and when the tuning should be performed
  - Request the execution of application modifications
- **Perform application tuning online**
  - Generate or load tuning code to individual application processes
  - Perform modifications
  - Synchronize the modifications with the application execution to ensure the correct functioning of the application
  - Undo tuning when considered necessary
- **Evaluate the profitability of performed tuning**
  - Measure the performance monitoring cost
  - Measure the tuning cost
  - Evaluate the impact of modifications on the overall application performance

5.4. Design issues

The following statements explain the thinking behind the decisions taken during the design of our tuning environment. In order to build an efficient environment that provides all required services for dynamic tuning, we had to consider carefully many aspects. Here we describe each of them indicating main decisions, problems and techniques that were considered while designing and implementing them.

5.4.1. Control of the execution of the parallel application

As indicated previously, the current version of our environment is dedicated to PVM-based parallel applications. The PVM application consists of several tasks (processes) that cooperate to solve a common problem. These tasks are distributed on a set of machines that form a *virtual machine*. To provide a communication service between all the tasks, PVM runs on each machine a process called PVM daemon (pvmd). The daemon controls the creation of tasks and their communication on the machine it is running and it also communicates with other daemons. Figure 5.1 shows an example of the distribution of application tasks in a PVM virtual machine.

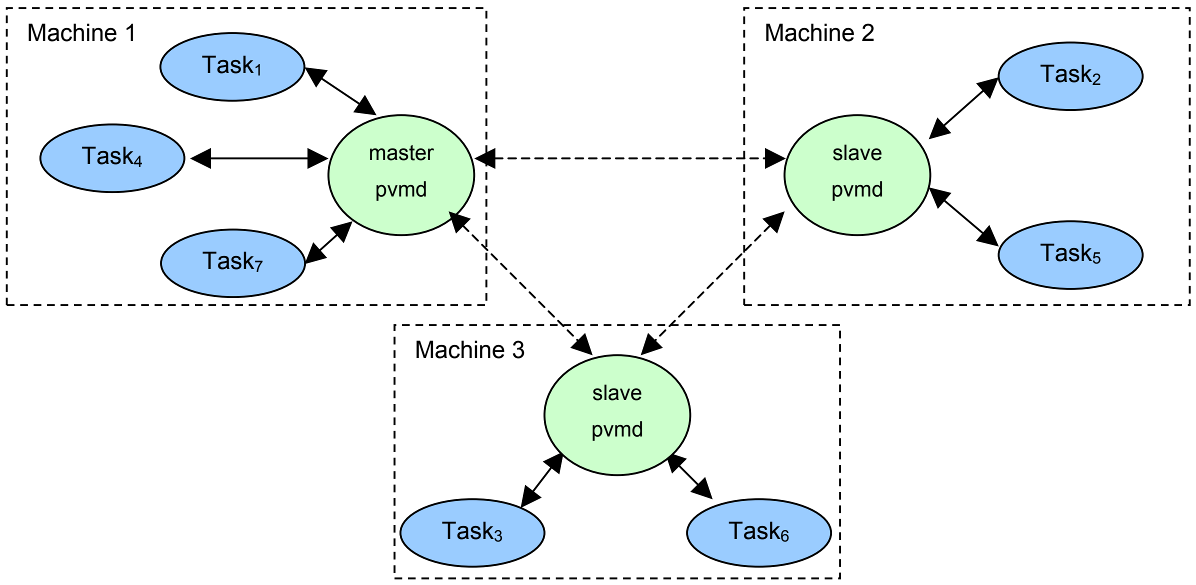


Fig. 5.1. Distribution of application tasks in the PVM virtual machine.

PVM daemons form Master/Slave structure. For each machine one PVM daemon is executed. If PVM is running, there is always one master daemon executed as a first daemon in the virtual machine, the rest of daemons serve as slaves. Master daemon controls the whole virtual machine. When a new machine is to be added, a new slave daemon must be created on that machine. This request comes to master daemon and it runs a new slave daemon on a specified machine. When application spawns a new process, appropriate request comes to the master daemon. Then master daemon creates the process on a local machine or redirects this request to remote slave daemon if process is to be created on a remote machine. PVM spawns new processes on all available machines of the virtual machine using the round robin technique (a circular queue is kept).

If virtual machine already contains a number of machines or a new machine is added dynamically to the virtual machine, it is obvious that PVM may run processes there. Therefore, to control the execution of the PVM application, our tuning environment must take control over all processes on all these machines where they are running. To provide these capabilities the tuning system must control the creation of a new PVM task and the start of a new PVM slave daemon. In order to take control over the PVM application, there are two main problems to be resolved. First, the tuning system must use two special services provided by PVM, namely *tasker* and *hoster*. Next, it is necessary to distribute the modules of the tuning environment to all machines where the tasks are running. Therefore we created the **Application Controller (AC)** program that is responsible for controlling the execution of the parallel application by means of the tasker and hoster services. This program is automatically executed on all machines that form the virtual machine.

The tasker service allows the Application Controller to receive the request when a new PVM process must be created on a given machine. A PVM daemon is exempted from the process creation; all necessary steps must be performed by the AC. The hoster service handles the creation of a new PVM daemon on a remote machine. This service allows the AC to receive the request when a new PVM daemon must be created. Master PVM daemon is relieved from the slave daemon creation duty; all tasks required to create a new daemon must be done by the AC.

Each instance of the Application Controller provides the tasker service what permits to control a new process creation on the local machine. When starting a new process on a local machine, we are supported by the DynInst library. In order to create a new application process, we use a special method of the library that automatically takes care of the process creation phase. This solution is very reasonable, because via DynInst we have the process control and we can easily monitor and tune this process inserting and removing the instrumentation code.

To support a creation of a new application process on a remote machine, the Application Controller must be able to distribute itself to the remote machines. However only a single AC can run the hoster service (there might be only one hoster in PVM). Therefore we must identify the **Master Application Controller** that runs the hoster and a set of **Slave Application Controllers**. The first executed AC is considered the master and it controls the distribution of Slave ACs as well as the local task creation. Slave ACs are distributed all over the PVM virtual machine (one process per host) and each one supports tasker service controlling all the application tasks on its local machine.

All these design decisions are dedicated to PVM-based applications. Another library, such as MPI 1.0 does not provide dynamic task creation. An application is static since no processes can be added to or deleted from an application after it has been started. Therefore in this case the application control gets much simpler. New functionalities supported by MPI 2.0 consider dynamically created tasks, but this is out of the scope of our work.

5.4.2. Performance monitoring

The performance monitoring is responsible for the instrumentation of a parallel application and collection of information about application behavior. The **application instrumentation** and **data collection** must be done dynamically during run time without accessing the application source code. As stated previously, the performance monitoring is based on the DynInst library that enables the application address space manipulations and dynamic insertion and removal of monitoring code. We have also mentioned before, that we distinguished in our environment the Application Controller that is responsible for controlling the PVM application execution. Since this program is distributed and already has access to each individual PVM application task (it creates all tasks via DynInst library), we decided to include in it the performance monitoring module as well. The Application Controller uses DynInst to generate the appropriate monitoring code (i.e. snippets). Then during run time, the AC inserts the snippets into or remove them from the running task. In this way all tasks of the parallel application can be monitored.

One of the principal goals of the performance monitoring is to provide an information about the application execution during run time. To collect such an information, the application must be instrumented. The instrumentation must be inserted into the original program execution at points needed to detect performance problems (concerning the application knowledge our environment is based on a.k.a. measure points), then it must generate necessary information and finally, this information must be sent for analysis. We decided to drive dynamic monitoring basing on **event tracing.** There are two basic types of events:

- Entry of function call – an event is generated when the execution reaches an entry of a particular function. Typically, the selected function parameters are attached as event parameters.
- Exit of function call – an event is generated before the function returns the execution to its caller. Typically, this event is not associated with additional parameters. Instead it is used to measure execution time of a function.

Optionally, other types of events may be considered, for example block-level events such as loop entry or exit.

This event tracing technique is frequently considered invasive one since large amount of data can be produced. However, in the same time it is the most precise and the most flexible as events contain the detailed information about what happened, when, where and in which circumstances. Our environment inserts the instrumentation only when needed. This allows for provision of the precise information about the application behavior, but at the same time it also controls the intrusion introduced into the application and network. The complementary solution that could allow for minimizing the intrusion is profiling. It allows one to periodically obtain the statistical (aggregated) information about selected performance metrics and hence significantly reduce the amount of information to be transferred. In our work we decided to focus on the event collection technique, leaving the dynamically insertable metrics as possible extension.

Event tracing requires the precise definition of data associated with an event. Typically, each recorded event includes a set of attributes:

- **What** – what action occurred (event identifier or/and function name)
- **When** – the time when the event occurred (timestamp)
- **Where** – the location where the event occurred (e.g. host, task, line number, source file name)
- **Parameters** – additional event-type dependent custom parameters (e.g. function parameter)

Taking advantage of the experience gained from working with available monitoring tools, we intended to create generic **event representation format** that includes all mentioned attributes. It must be pointed out that events collected in a similar format can be widely applied by other analysis tools. We have already discussed that some of the monitoring tools generate events in the PICL format and there are analysis tools that take as input trace files in this format. Therefore, we designed the event format to be easily adapted to the PICL format.

As we have indicated previously, the online monitoring consists of two principal phases: instrumentation and collection. Our design assumption is event-based monitoring and analysis and hence the instrumentation must generate events that happen during the execution and deliver them for analysis. The application instrumentation includes generation of code to be inserted and insertion of this code. In the collection phase, there is gathering of events generated by inserted instrumentation code, and delivering them for analysis. The performance monitoring is based on the DynInst library that provides the possibilities to create an instrumentation code and manage it: insert into or remove from the specified points manipulating on the running process. By means of DynInst library, the monitoring service is then able to manage the instrumentation that will generate the information about the application execution.

For example, if a function `foo()` is invoked, appropriate event with all necessary information that we indicated previously must be generated and sent for analysis. To instrument an application via DynInst we have to create in the monitoring module a special code called snippet that will be able to collect all necessary information from the specified function. Next the collected event data is delivered for analysis. Then during run time the snippet will be inserted into the memory of the running process at all specified point/points that are needed to discover performance problems (e.g. entry of the function `foo()`). Each time when the function `foo()` is invoked and a location with inserted snippet is reached, the snippet code will be performed. However, creation of such snippets is not a trivial task, since basically, creation of snippets is done by means of the AST (see Chapter 4). The AST for this kind of snippets is quite complex and hard to manage. Therefore, we use a special service provided by the DynInst library that facilitates snippet creation.

DynInst library provides the method to **load library dynamically to the application** (mutatee) process. This dynamic library is attached to the memory of the process during the execution. The loading process of the monitoring library is shown on Figure 5.2. Instead of writing complex snippets based on AST that generate events, we have the possibility to write the code in C language. In the library we implement functions to be performed when event is being generated during the application monitoring, namely collect event data and deliver it for analysis. The code of the monitoring module provides only creation of snippets that are build as AST, but are simple. This kind of snippet only invokes the call/calls to the appropriate C function/functions from the run time monitoring library passing to it/them all necessary parameters taken from the function the snippet is inserted. Dynamic library is loaded for each process separately, it means that each application process has its own copy of the dynamic library in the memory address space.

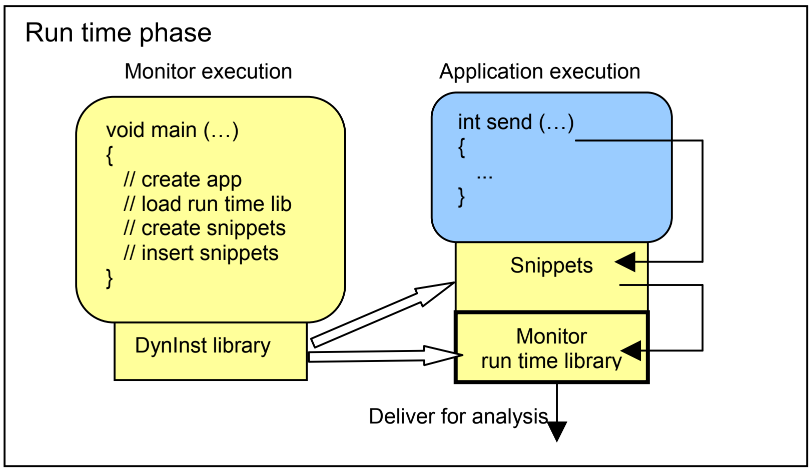


Fig. 5.2. Monitoring run time library loaded into the memory of the running process.

Application distribution and event-based performance analysis cause important problem, namely **clock differences** of a set of machines. In order to analyze the application behavior, in some circumstances, it is required to provide events with precise, global clock information (i.e. casual ordering). Therefore, events are annotated with timestamps generated on the time references. Virtual machine contains a set of machines and it is not ensure that their clocks are exactly the same. If clocks are different on different machines, event timestamps can differ significantly, and hence the analysis will not be correct. Each process performs its work and inserted instrumentation generates events. Events of each process are annotated with timestamp from the local machine, and although they are ordered for the process (partial event order), they can be globally unordered for the whole application. The analysis may require the casual event ordering (e.g. the send ends before the associated receive ends) and therefore the events must be preprocessed. This problem is illustrated on Figure 5.3. Generally, when event timestamps are not global time referenced, the ordering operation might be complex because of the determination of correct event order and the need for timestamps modifications after the event generation.

We took the problem of time differences into consideration and we decided to provide global timestamps approach. In this case all events must be annotated with timestamps referenced to one machine. To provide such a timestamp the clock synchronization must be performed. All machines must agree on time and synchronize their local time with a time on a reference machine. After this operation events can contain global timestamp and hence are time coherent.

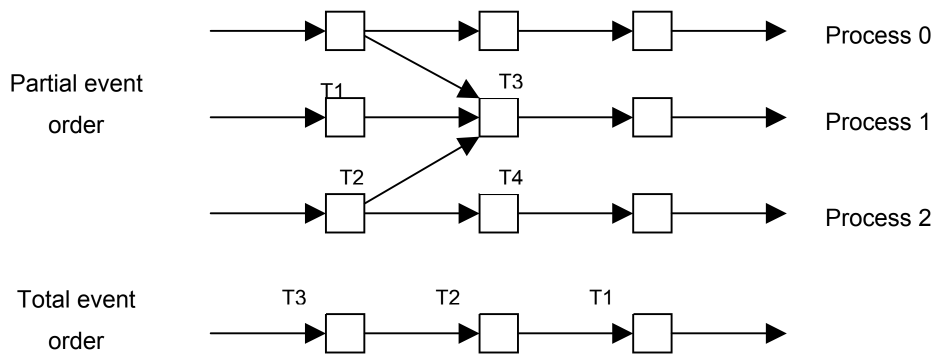


Fig. 5.3. Partial and total event order.

Our environment distributes the Application Controller all over the PVM virtual machine and since it resides on each machine, it has access to the local machine clock. Therefore, this program can be also responsible for the **clock synchronization**. The synchronization performed by the Application Controller is illustrated on Figure 5.4. In order to provide the synchronization service, we chose as a referenced machine the one where the Master AC is running. The Master AC runs the separate time server process that waits for the requests. Each Slave AC must then synchronize the local clock with the master machine. Therefore, once the Slave AC starts on a remote machine, it sends requests to the time server. Then the Slave AC calculates and stores the clock difference. When an event is generated, it is logged automatically with the correct timestamp (adding/subtracting the difference to/from the local timestamp) and hence the clock desynchronization is minimized. Therefore, when the events are sent for the analysis module, it will receive events with adjusted timestamp. Then by applying casual ordering the analyzer can achieve the required ordering level.

Clock synchronization is a very complex issue and it is difficult to implement it efficiently and reliably [Par98]. The main problem is to provide precisely defined difference between clocks of two machines. Many aspects must be taken into consideration when determining the exact clock difference, for example network load, time required to send and receive the time through the network, processing of the time request, etc. The well known and effective, but complex solution is described in [Rab97]. This method is used in the Tape/PVM, the monitoring tool that we have presented in Chapter 2. In the current version of our environment, we use simple method of synchronization. Although the method is simple, it provides us with the well-approximated clock differences. Taking into account the existence of better method, the design of the AC is well prepared for the changes of the synchronization method. It can be easily adapted for the new, more efficient and reliable implementation of the clock synchronization.

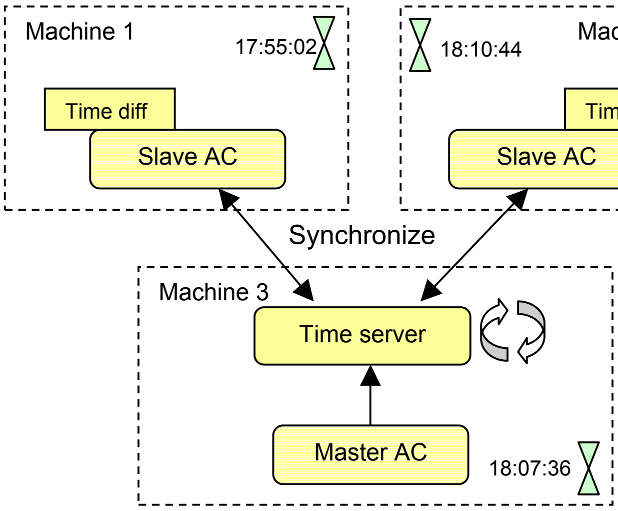


Fig. 5.4. Clock synchronization.

5.4.3. Performance analysis

The analysis is responsible for the automatic performance analysis of a parallel application "on the fly". It is able to examine application behavior, identify performance bottlenecks, and give concrete solutions that overcome these problems. In general, the analysis process must be performed in continuous parallel computational loop: application monitoring, core performance analysis in order to detect the problem and finally solving or minimizing the impact of the problem by applying tuning actions. The analysis process continues until the application terminates.

The analysis must be done globally with taking into consideration behavior of entire application. For this purpose we distinguished a distinct module that will be responsible for monitoring data collection and its analysis. We suppose the performance analysis to be time-consuming. To minimize the intrusion introduced into the application execution, the analysis module should be performed on a dedicated and distinct machine (the performance "optimizer" machine). In Chapter 4 we talked about the scalability of such solution and the possibility to perform some parts of analysis locally, but this is future work and this is out of the scope of the design.

The analysis defines and processes the performance measurements at run-time. The on-line analysis can focus on specific execution aspects (i.e. most severe problems), selectively refining its measurements in light of the previous results. Therefore, the instrumentation can be added or removed automatically according to the actual program behavior. The application can start with generic initial instrumentation and then this instrumentation can be selectively changed by requesting more or less detailed information. This leads to a reduction in the amount of measurement data.

Performance analysis (i.e. problem detection) can be performed using a number of different methods, for example analytical performance model, rules or probabilistic model. An analytical model describes behavior of an application and determines how to improve the current settings and in consequence how to find the optimal execution time. In this case, the performance model must contain a set of related tuning actions. The analysis based on such an analytical model and included mathematical formulas receives the monitoring data and calculates the actual and optimal setting. If it decides that some settings may be changed to improve the performance, an action should be applied on parameters related with these settings. Other analysis method is based on a set of rules. During the performance analysis, the input monitoring data is applied to the rules. If the result is positive, then an appropriate action must be invoked. In this sense, such method must contain a set of related tuning actions, as well. There is also other analysis method based on models, but in this case this is the probabilistic model. In such solution, a parameter under control is adapted by calculating its value space using heuristic algorithm. This method is a method of probes and errors. The parameter is set to a new value and it is continuously controlled if the applied change was successful or only make the performance worse.

Each analysis method has its advantages and disadvantages, it depends on a particular case which one is better. We want to investigate many model and not limit us to only one. Therefore, we have to provide the tuning environment easy to extend and to apply different analysis methods. However, for the purposes of this work we concentrate on the analysis based on the performance model.

Each performance model contains a set of input data that it requires for calculations, as well as a set of output data that represents how the application should be changed to improve its performance. Moreover, in Chapter 4 we described that tuning system, to be possible and effective, must be provided with the application knowledge. We correlated these two issues and we based the analysis on application knowledge of possible problems and their solutions. We specified the knowledge as measure points, performance model (as formulas and conditions) and tuning points/action/synchronization.

The knowledge that is provided to our environments contains a set of different problems that can be monitored, detected and solved. All required information and processing related to each problem is called **tuning technique**. To support the analysis of many problems, the tuning system includes the catalog of tuning techniques where each technique solves a particular problem. Initially, we decided to start with an approach that treats all provided techniques separately. The analysis simply runs a number of techniques simultaneously and when a particular technique detects the problem it activates the tuning. In future it would be interesting to investigate another approach as for example analysis based on hierarchy of problems. In this case the tuning system has a hierarchical catalog of performance problems where each problem is associated with an optimization technique. Such approach of problem searching and refining is used in Paradyn [Hol93].

From the one side, we decided that a tuning technique can be provided by our tuning environment or by the developer that utilizes our environment. First provision method will support black box approach of tuning since there is no intervention of a developer side. Second one will be cooperative tuning as developers must prepare all required information and adapt their application to be aware of possible dynamic modifications. Although we distinguished two approaches, we want our environment to support one and uniform knowledge inclusion mechanism. Therefore, we looked for a one solution of tuning technique provision adequate for both of them.

From the other side, an application knowledge cannot be hard coded within the tuning environment, because the system would not be easy to extend. The environment must provide the possibility to add new tuning techniques. One of the good solution would be to declare such a technique externally (using a declarative language) and then interpret it. However, performance models can have heterogeneous forms (e.g. mathematical formulas, complex conditions) and it is really difficult to declare them. Therefore, the most flexible solution is to provide a tuning technique using programming language. To add such a technique to the environment we need a kind of component that is easily loadable.

For both reasons, we decided to provide an external, compact module in the form of dynamically loaded shared library. One library represents one particular performance bottleneck and is able to cooperate with the tuning environment. Such a library is called **tunlet**. Each tunlet contains specific information related to a bottleneck that can occur in the application. A single tunlet addresses one concrete performance problem by implementing a particular tuning technique. To be able to cooperate with the environment, the tunlet implementation is based on the Dynamic Tuning API provided by the analysis module.

A tunlet must be prepared for a particular problem. All necessary information is determined by the investigation of the possible bottlenecks in the operating system, problem-specific library implementation or application. The tunlet must decide what is needed to detect a problem (measure points) and what must be changed to improve the performance (tuning points/action/synchronization). To perform the analysis it must receive meta data about the application model and monitoring data generated by the instrumentation. The tunlet hence must cooperate with the analysis service of our environment. Therefore, we decided to include an API for the tunlets. The analysis module of our environment must provide this API, its implementation carried out communication with the rest modules of our environment and finally a container of the tunlets. Figure 5.5 shows the performance analysis based on the catalog of tuning techniques implemented in the form of tunlets.

The tunlet must provide the analysis with the measure points that represents what should be instrument in the application in order to find a bottleneck. The analysis service then broadcasts required measure points to all ACs modules responsible for the application performance monitoring. When an application is running, the searching-bottleneck phase starts. The analysis service is responsible to continuously collect events generated by different processes. When a meta data of the application model or event records come they are sent to the appropriate tunlet for analysis. The tunlet evaluates the performance model using collected event records and checks if a bottleneck occurs. If it is the case, it finds what should be changed and its optimal settings. It also decides if the changes must be performed in the whole application or in the particular task/tasks.

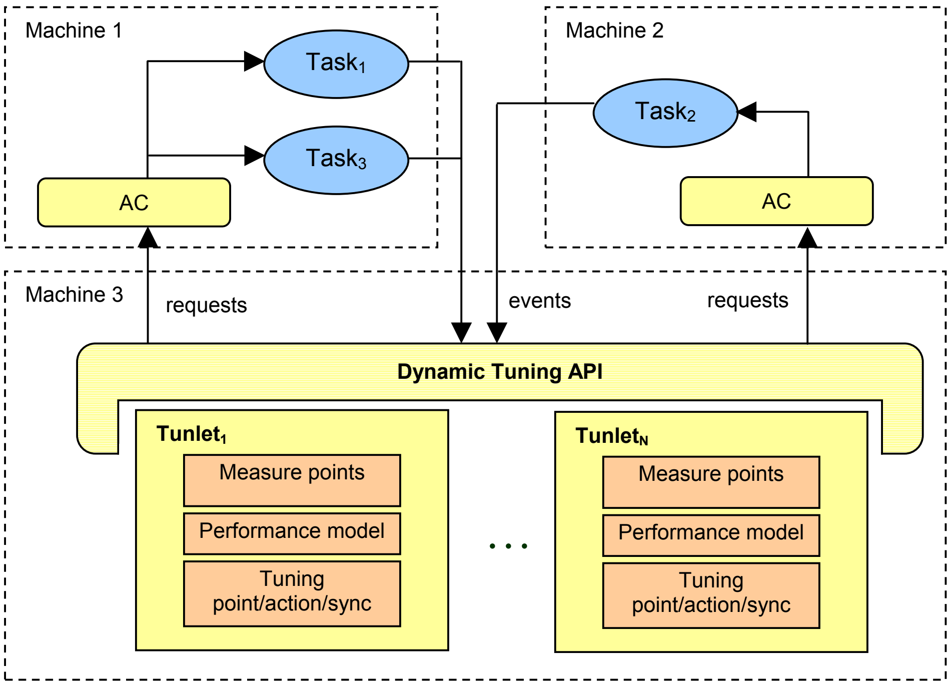


Fig. 5.5. Performance analysis based on knowledge provided as tunlets.

For example, when considering performance inefficiencies, we can define that the efficiency of a process is considered as the percentage of time that it is doing useful work. The analysis can search those intervals where processes are not doing any useful work (they are simply blocked, waiting for a message). Concerning PVM applications, we can measure it inserting the instrumentation into entry and exit of function pvm_recv(). The idle time is calculated as the difference time between the entry and exit. When idle-time intervals exceed the limits of threshold values, these intervals should be minimized in order to improve the performance of the application. If the evaluated behavior is not considered satisfactory, its causes are determined. A cause may be, for example, non-optimal work size being assigned to slave processes. The model is then used to calculate the optimal settings and to decide what tuning actions should be performed.

Finally, the tunlet searches for the appropriate tuning modifications that should be invoked in the particular process or in the whole application. It notifies an analysis service, which in turn sends appropriate request to the AC, that a given tuning action should be invoked at a given point in a given process. The information how to synchronize the changes with the process execution is passed as well. For example, in a particular Master-Worker application, if the analysis determines that a number of worker parameter should be changed in a master process, the following information is sent to the AC: the process identifier, the name of a variable together with its new value, and synchronization point.

The analysis process continues until the application terminates. Obviously, all this time the monitoring data is sent for analysis. In certain cases, the tunlet may need more information about program execution to determine the causes of a particular problem. In other cases when the problem has been already detected, the tunlet may not need any more monitoring data. It can therefore request the analysis service (that in turn sends appropriate request to AC) to change the instrumentation dynamically, depending on the necessity to detect performance problems. Dynamic changes of monitoring data can decrease the intrusion introduced into the application execution.

5.4.4. Tuning

The performance tuning is responsible for automatic modifications of a running parallel application. It is based on decisions given by the performance analysis. When a problem has been detected and the solution has been indicated by the analysis, the performance tuning service receives the solution and automatically applies it changing the running task. The application of solution is done by means of DynInst library since this operation must be done during run time without source code, recompilation and rerunning the program.

When applying specified code modifications, the task memory is manipulated by invoking appropriate changes. Therefore, access to the corresponding task is required and hence performance tuning service must be distributed all over the PVM virtual machine where application tasks are running. As stated previously, we distinguished the Application Controller responsible for controlling the PVM application execution. Since this program has access to each individual PVM application task, we decided to include in the AC the performance tuning module as well.

The tuning methods must be kept simple and well determined since all the changes performed on an application cannot affect its correct functioning. When changing running application, the tuning module must be provided with exact information about what to change, where and finally when. Therefore, the solution to be applied that comes from the analysis contains all required information, namely a target task, tuning action (what), tuning point (where), and synchronization (when). Moreover, in Chapter 4 we distinguished the set of possible modifications that concern DynInst library functionality. Therefore, we carefully determined tuning actions that can be applied on the running process. We decided to include in the performance tuning a set of predefined modifications that are performed on tuning points and can be activated by the performance analysis.

We consider the following tuning actions:

- function replacement – replaces all calls to one function with calls to a new one. The implementation of a new function must already reside in the application memory.
- function invocation – inserts a new function invocation code with a specified attributes at a given location. The implementation of a new function must already reside in the application memory.
- one time function invocation – inserts a new function invocation code with a specified attributes and invokes it only once. The implementation of a new function must already reside in the application memory.
- function call elimination – removes calls to a specified function from a given point.
- function parameter changes – sets the value of an input parameter of a specified function.
- variable value changes – modifies a value of a specified variable.

When one of these actions is activated, the performance tuning module generates the corresponding instrumentation. Then it inserts the generated code at the specified point and if necessary synchronizes its invocation with the application. The tuning instrumentation can be generated in two ways:

- a snippet code directly calls appropriate DynInst library method to invoke the tuning action – such a snippet is simply and does not require to perform many operations. For example, in the case of the I/O bottlenecks caused by flushing, the tuning action can eliminate the flush() function call. To perform this action a snippet simply invokes `BPath_thread::removeFunctionCall()` method from DynInst library.
- a snippet code invokes appropriate method from the run time tuning library – a snippet is created as in the case of monitoring instrumentation and it simply calls a function from the run time tuning library previously loaded to the process memory. A snippet is simple but it requires a code of additional function provided by the loaded library. For

example, when the analysis module decides to incorporate custom memory allocator, a new allocation function must be called. The tuning action then replaces all calls to the standard function e.g. `malloc()` with a call to the new one. The new function must reside in the process memory and hence is provided in the run time tuning library. To perform this action a snippet invokes `BPath_thread::replaceFunction()` method from DynInst library.

The changes made by performance tuning module 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 might appear to be a constraint. However, as it has been already pointed out, the main performance problems of parallel distributed application are those that appear many times during the execution of the application.

5.4.5. Overhead minimization

One of the fundamental issues for the tuning system in achieving the profitability is the minimization of intrusion. Obviously the tuning system must guarantee that the intrusion is much smaller than expected benefits. Therefore, it is necessary to minimize the overhead introduced by the existence and functioning of all the components of the tuning system. To achieve that we must take into consideration some precautions such as intrusion preventive actions, efficient implementation techniques, and code optimizations.

The first source of intrusion is the necessity of distributing processes of the tuning system to all computers when the application processes are running. Therefore we designed the Monitor/Tuner process to be small and resource conservative. Moreover, as described in the previous section, there is only one process running per host. In the normal conditions, the process remains idle and does not affect the functioning of the application processes until there is a request to be served.

The next source of intrusion is the requirement to manage the startup/exit of each individual application process. This is performed by means of DynInst API and requires the image of the application process to be parsed before the process is executed. Typically this operation takes about several seconds (in function of the image size). Although this operation can be considered irrelevant when performed before application start, the cost associated with frequent task creation during application execution may be more significant.

Next, each modification of the application process triggered by the necessity to insert or remove monitoring or tuning code comes at a cost. In particular each dynamic instrumentation requires the temporal suspense of the application execution, the completion of modifications, and the restoration of the execution. Naturally, the new code inserted into the application causes additional overhead that mainly depends on the complexity of the introduced code. Therefore it is important to keep the instrumentation simple.

The principle of the dynamic instrumentation is that the instrumentation code needs only to reside in running application as long as it is needed to gather data. So the idea is to insert it as late as possible, and remove it as soon as possible. Therefore, in our tuning system the instrumentation can be added or removed automatically according to the actual program behavior. Instrumentation can be inserted only when a specific performance problem in the analyzed application is suspected. For example, the running program can be measured with an initial set of instrumentation. Next, when some thresholds are exceeded, an additional instrumentation might be introduced to obtain more detailed information. Finally, when the problem has been solved, the required measurements can be reduced or even removed. The instrumentation overhead can therefore be dynamically reduced and controlled. We avoid the constant overhead as it would be in the case of classical profiling that requires all the functions to be instrumented during the entire execution of the application.

The next source of intrusion results from the need to have a global application view to perform the global analysis. Due to physical distribution of modules of the tuning system, it is necessary to intercommunicate them. This may affect the network performance and hence we need to minimize the frequency and the number of exchanged messages. The communication between modules of our tool cannot delay the communication between tasks of the tuned application. For example, in the case of the monitoring module, we decided to use compact, binary message format and event buffering mechanism to reduce the number of generated messages.

Finally, the performance analysis process cannot be too expensive. It should be lightweight and not computationally intensive. Moreover it must react to changes in the application behavior and make adequate decisions in a timely manner (in required short time-frames). Therefore we consider important to keep the analysis process simple.

5.4.6. Portability

Considering hardware available in our laboratories, we decided to implement our tuning environment for SPARC Sun Solaris platform. However, as we have mentioned the environment does not have any platform-specific dependencies and is kept compatible with POSIX standard. Our tuning system depends on PVM and DynInst, but both libraries are available for many different platforms (UNIX, Windows). Therefore, the environment can be easily ported to different platforms.

5.4.7. Support for alternative communication libraries

The current version of our dynamic tuning environment is implemented in C++ language and is dedicated to PVM-based applications. However, it can be adapted to support applications that use other message passing communication libraries. The most significant dependencies are related to the application control and to communication library-dependent tuning techniques. To make the environment extensible and easy to maintain, it has been designed using object-oriented methodology and we have intended to simplify possible extensions providing the code reusability. Therefore, to provide for example MPI support, only selected, isolated classes should be reimplemented and others (such as tasker, hoster services) disabled. Some of the tuning techniques are dedicated to concrete libraries, e.g. PVM and they cannot be used with MPI applications. It would necessary to identify other MPI-specific tuning techniques and implement them as tunlets.

5.5. MATE

We propose a novel environment called **MATE** (Monitoring, Analysis and Tuning Environment) that enables dynamic performance improvement of distributed parallel applications. MATE supports three basic functionalities: performance monitoring, performance analysis and tuning. All these phases are performed automatically and continuously “on the fly” by our environment.

5.5.1. Architecture

Basically, MATE consists of the following main components that cooperate among themselves, controlling and trying to improve the execution of the application:

- **Application Controller** (AC) – a daemon-like process that control the execution and dynamic instrumentation of individual PVM tasks.
- **Dynamic monitoring library** (DMLib) – a shared library that is dynamically loaded into application tasks to facilitate the performance monitoring and data collection.
- **Analyzer** – a process that carries out the application performance analysis and decides on monitoring and tuning.

Figure 5.6 presents the MATE architecture in sample PVM scenario. In this example the PVM application consists of 3 tasks distributed on 2 different machines. To start the application, MATE distributes the Application Controller processes to both machines to control the startup of the tasks. There is one Master AC residing on the machine where master PVM daemon is running. The Master AC provides the virtual machine control (i.e. dynamic addition of machines to PVM virtual machine), local tasks creation and moreover serves as the time server for clock synchronization between hosts. The Slave AC runs on another machine. It controls the creation of local tasks and synchronizes the clock with the master AC. In order to control the creation of tasks, both ACs communicate with the local PVM daemon (pvmd). When a new PVM task is started, the AC loads the shared

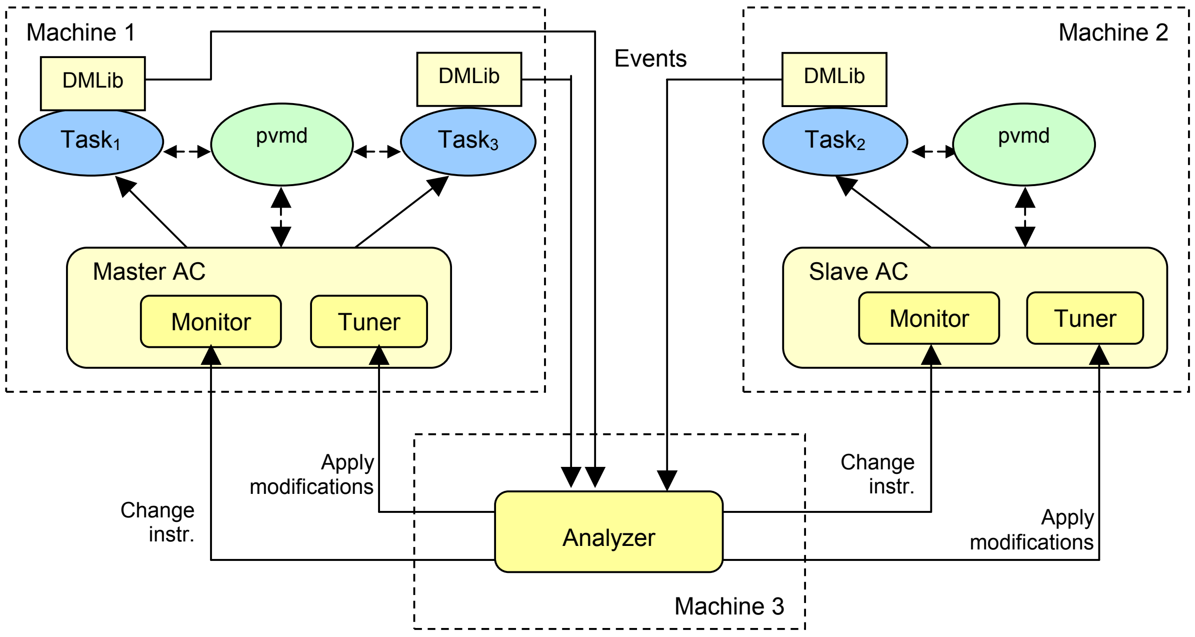


Fig. 5.6. Architecture of the MATE dynamic tuning for PVM.

monitoring library (DMLib) to the task memory that allows for its instrumentation. During execution, the ACs manages the instrumentation of each task. This allows the Analyzer to dynamically add/remove events to be traced and apply tuning actions. The shared monitoring libraries are responsible for delivering registered events directly to the Analyzer.

In the following sections we describe with details all modules of the MATE environment. We present their functionality, interfaces and limitations.

5.5.2. Application Controller

As introduced previously, each Application Controller process manages a set of PVM tasks running on its local machine. This process provides the following services:

- Distributed application control
    - Startup/exit of PVM tasks
    - Startup/exit of new PVM daemons and slave ACs
    - Clock synchronization between ACs on different hosts
- Application instrumentation management
    - Manage instrumentation of running tasks
    - Allow the Analyzer to remotely add/remove instrumentation
- Performance monitoring
    - Load shared monitoring library into application task
    - Generate monitoring snippets
    - Insert/remove the snippets
- Performance tuning
    - Load shared tuning library into application task
    - Generate tuning snippets
    - Insert/remove the snippets

The Application Controller consists of a number of cooperating modules. Figure 5.7 presents the internal architecture of the Application Controller program and the following paragraphs describe in detail each module.

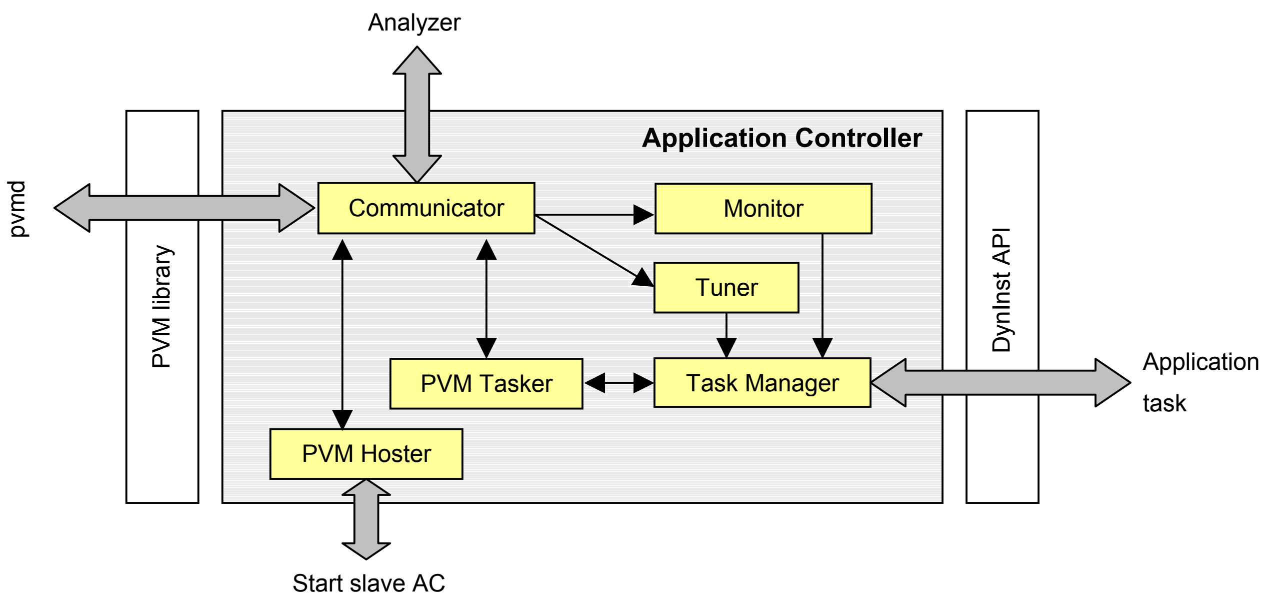


Fig. 5.7 Internal architecture of the Application Controller.

Communicator

This is the central module of the AC that handles the communication with external world using TCP/IP protocol. This enables the AC to process the Analyzer requests (i.e. add/remove monitoring instrumentation or apply tuning actions) and handle PVM notifications (i.e. spawn new PVM task or add new slave host). The Communicator module dispatches the incoming messages to be processed by appropriate modules.

Because the communicator must be able to handle both PVM and Analyzer communication simultaneously, we have chosen an object-oriented design pattern called Reactor [Gam95] to implement this module. This design pattern handles service requests that are delivered concurrently to an application by one or more clients. This pattern can be utilized for applications that can receive simultaneously many requests from different clients. Reactor allows the application to wait for incoming requests, demultiplex them and finally dispatch them to the corresponding handlers. For each request application offers a provider that handles the request. The reactor implementation uses low-level `select()` system call. When a message arrives, it is demultiplexed and in function of its type it is then dispatched to the appropriate AC module. The module receives the message and performs corresponding operations. For example, when PVM message (`SM_STTASK`) arrives, the reactor dispatches the message to the PVM tasker module that in turn creates a new PVM task.

PVM tasker

According to the PVM foundations, each of the processes of the parallel application while performing its computations can spawn new processes. When a PVM task wants to spawn a new task, calls the `pvm_spawn()` function that in turn communicates with the PVM daemon and sends it the appropriate request. The local daemon handles the request by physically starting a new process. To change the standard behavior of the process creation, the AC provides the module that implements the PVM tasker service.

The functioning of this module is illustrated on Figure 5.8. The PVM tasker service connects to the local pvmd and registers itself as a tasker (1). After the registration, the AC can start the parallel application by creating the application "father" task (2). The father task usually spawns additional tasks. Therefore, to control their creation, the tasker registration must be performed before father task starts. When the father is created, the AC inserts the appropriate instrumentation and then allows the process to execute (3). When the father process wants to create a new child process, it sends spawn request to a local PVM daemon (4). The PVM daemon receives the request, checks if there is a registered tasker and forwards the spawn message `SM_STTASK` to the tasker that becomes responsible for a new process creation (5). Next, the tasker creates a new child process extracting all necessary parameters from the received message (6) and delegating the creation to the Task Manager module that actually creates the process and automatically inserts the required instrumentation. (7). Next, the newly created process starts and registers itself with the PVM daemon as a PVM-task and communicates without any changes with a father process via PVM daemon. Finally, the tasker duty is to notify the PVM daemon about the termination of the child process. This is performed by sending the message `SM_TASKX` to the pvmd.

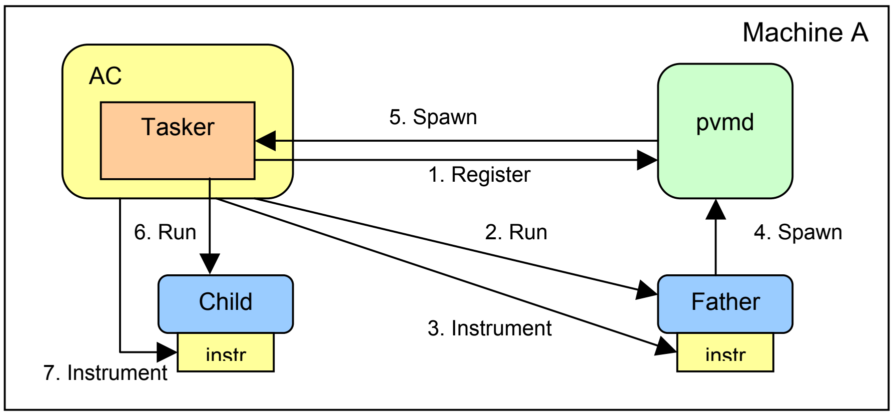


Fig. 5.8. An example scenario: AC starts a parallel application.

PVM hoster

In the PVM, when the father process spawns child processes, they can be distributed all over the virtual machine (VM). The VM can be configured before the application startup, but it can also grow dynamically during the parallel application execution. The AC goal is to monitor all the processes of the application, hence it must be able to control the VM as well. Therefore, the AC must perform two jobs. First, during startup it must check the existing configuration of the VM and distribute itself to all hosts the VM consists of before the monitored application launching. It is illustrated in Figure 5.9. Second, it must take control over the creation and removal of hosts from the VM since during the monitored application execution a request to run/remove a remote PVM daemon may appear. In this situation Master Application Controller must implement the PVM hoster service and launch a new Slave Application Controller on the indicated remote machine together with a new PVM daemon. This case is presented in Figure 5.10. Only the solution that we have just presented supports the control of all PVM processes on all machines.

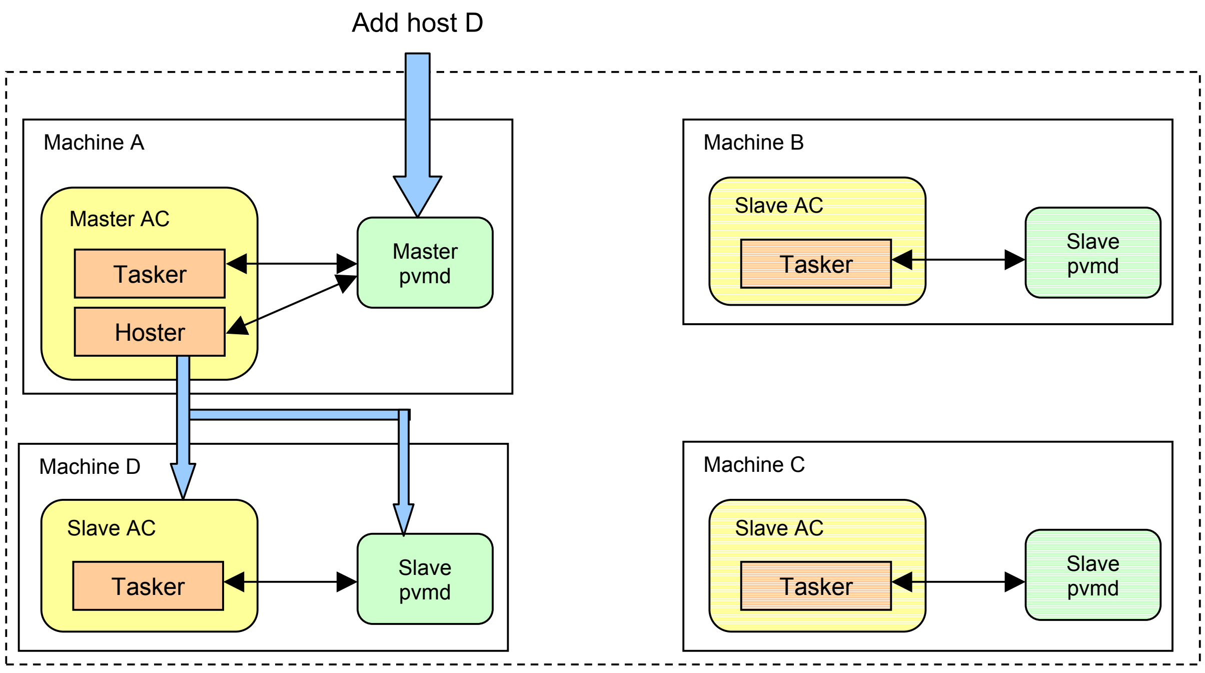


Fig. 5.9. Running a new Slave AC process when a new host is added dynamically to the PVM virtual machine during the monitored application execution.

In general, a new machine can be added to the PVM virtual machine in two ways: programmatically by the application by means of API functions (i.e. `pvm_addhosts()`, `pvm_delhosts()`) or manually by the user who configures the VM from the PVM console. When the new host must be added, a request is sent to the master PVM daemon. The pvmd physically adds the new remote machine and starts the slave PVM daemon there. From this moment on, all task scheduling requests will also consider the new virtual machine configuration.

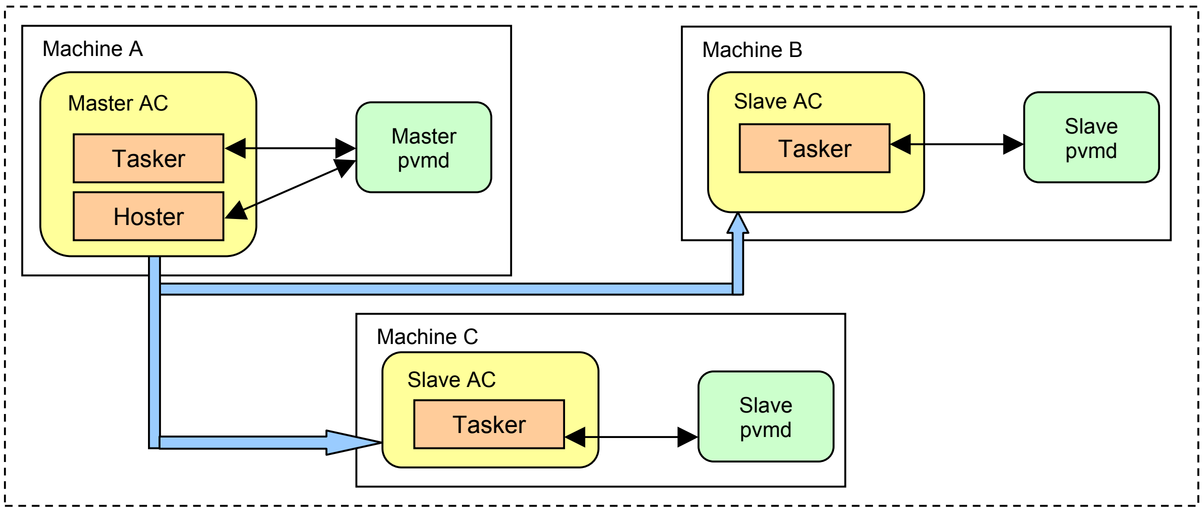


Fig. 5.10. Running a new Slave Application Controller process on all machines of the PVM virtual machine before the monitored application startup.

To customize this standard behavior of PVM the AC implements PVM hoster service. The functioning of the hoster service is illustrated on Figure 5.11. The PVM virtual machine can have only one hoster. Therefore only the Master AC runs this service and must register with master pvmd (1). When master pvmd receives a request to create a new PVM daemon (2), it checks if there is a registered hoster and if it is the case, it forwards the `SM_STHOST` message to the hoster (3). Next, the hoster extracts all necessary parameters from the message and starts a new slave PVM daemon process on an indicated remote machine (4). In order to monitor the application processes that can be spawned in the future on the

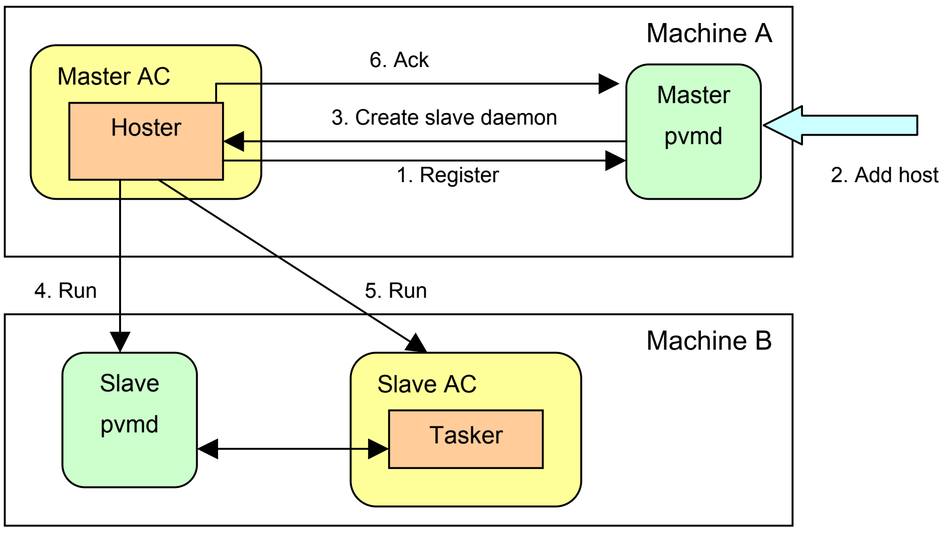


Fig. 5.11. PVM hoster service creates a slave pvmd and slave AC.

newly added machine, the hoster creates also a Slave AC process on that machine (5). The newly started Slave AC does not start the hoster service, but starts the tasker service. Finally, the hoster must send to the waiting master pvmd the `SM_STHOSTACK` message (6) that indicates the status of the slave daemon creation.

The remote creation of both slave pvmd and Slave AC is complicated by the possible hazards. Master AC must notify the master pvmd that the slave daemon was successfully started. However, it cannot be done before the Slave AC is completely started and registered as the tasker. Otherwise, the master pvmd could immediately spawn a new process on a remote machine and the Slave AC would not be able to control new task. To solve this problem, we have implemented a new program called Starter. This program is responsible for starting first the PVM daemon process, and next the Slave AC process. So when the add host request arrives to the Master AC, it launches the Starter process remotely by means of the `rsh()` command. In continuation, the starter creates both processes and waits for the Slave AC to confirm that it has registered the tasker service. Next, the `Starter` acknowledge to the Master AC that the whole process terminated successfully. Finally, the Master AC can notify the master pvmd without any hazards.

Task Manager

This module is responsible for the creation, management of inserted instrumentation and termination of application tasks. In order to manage all tasks on a local machine, the Task Manager (TM) must hold references to each created task. The TM uses DynInst library to perform the dynamic process creation and instrumentation.

When the TM is requested to create a new process (i.e. by the PVM tasker service), it uses the `BPatch_thread::createProcess()` method from DynInst API. When the process has been started, the TM loads a specified shared library to the application process. By default the MATE Dynamic Monitoring Library (DMLib) is loaded, but this can be customized if necessary. Next, any previously defined monitoring instrumentation is being inserted into the process to enable the Analyzer capturing the information from the very beginning (the details of the process instrumentation are explained later in this chapter).

The TM handles the insertion and removal of prepared instrumentation snippets. However, the particular code to be inserted is prepared by other module (i.e. Monitor and Tuner) and the role of the TM is to record the inserted snippets in order to handle their removal on request of other modules. The TM also handles the termination of tasks. When a task terminates, the TM is notified about this event by means of DynInst callback mechanism. The TM is then responsible to finalize the monitoring library. Next, the DynInst library automatically unloads the library and from the process memory. Finally, the TM must notify the tasker module that in turn notifies the PVM daemon that the process created by the tasker has terminated.

Monitor

This module is responsible for performance monitoring of the execution of a parallel application. As stated previously, currently the monitoring is based on the event tracing of function calls. The application is instrumented dynamically during run time and the inserted instrumentation generates events. Once the AC starts, the Monitor module receives from the Analyzer an initial set of events to be traced (these events are conceptually called measure points). At application process startup, the Monitor inserts the corresponding instrumentation code to record these events. Instrumentation can also vary on demand during run-time. If the Analyzer requires more or less information, it can notify the Monitor to change the instrumentation dynamically. Consequently, the Monitor supports modification of the set of monitored events, i.e., it is able to add new or to remove redundant events. The Monitor module offers the following API:

```
enum InstrPlace { ipFunctionEntry, ipFunctionExit };
enum AttrSource { asFuncParamValue, asVarValue, asFuncReturnValue,
                  asConstValue };
enum ValueType  { vtInteger, vtShort, vtCString, vtFloat, … };

struct Attribute
{
    AttrSource   source; // source of attribute value
    ValueType    type;   // type of attribute value
    char const * id;     // source-dependent object identifier
};

void AddEvent (int processId,
               eventId,
               char const * functionName,
               InstrPlace place,
               int nAttributes,
               Attribute * attrs);

void RemoveEvent (int processId, int eventId);
```

The API allows the Analyzer to dynamically add a new event to be traced by calling `AddEvent()` function. An individual event is defined as follows:

- processId – unique application process identifier
- eventId – unique value used to identify the event
- functionName – a name of the C/C++ function to be traced
- place – instrumentation place determines when the event should be generated: on function entry or exit.
- nAttributes – number of attributes that should be recorded with each event
- attrs – an array of Attribute structures that define each attribute to be recorded with the event. Each attribute can be either a global variable value (asVarValue), a parameter value of a called function (asFuncParamValue), a return value of another function (asFuncReturnValue) or a given constant value (asConstValue). The attribute has a value type (i.e. integer, float, etc.) and an identifier that identifies the variable name, function name or a function parameter index.

The previously added event can be removed during the application execution by calling `RemoveEvent()` function.

To perform dynamic event tracing, the Monitor uses DynInst library to insert the instrumentation code that generates events to be traced. For instance the instrumentation code may be inserted at the entry and/or the exit of `pvm_send()` and `pvm_recv()` functions, when it is necessary to monitor the network functions in order to find potential communication bottlenecks. To collect these events and deliver them to the Analyzer, the Monitor uses the dynamic monitoring library loaded into the task during its startup that communicates with the Analyzer using low-level event collection protocol based on TCP/IP.

To trace a new event, the Monitor builds dynamically an instrumentation code – so called snippet. The snippet collects all necessary event attributes by reading the value of function parameters (e.g. tid of a process the message is sent to) or global variables as defined by the event attributes. To read a value of parameter of instrumented function, the snippet uses DynInst class `BPatch_paramExpr`. Next, it invokes a recording function from the monitoring library (DMLib). The DMLib API is described in detail in the following sections. Finally, the snippet is inserted into the appropriate points of the process by means of DynInst `Bpatch_thread::insertSnippet()` method (e.g. to the entry of the function `pvm_send())`. Each time the instrumented function is executed, the inserted snippet code is invoked. The generated event is then passed to the DMLib and later delivered to the Analyzer.

Tuner

This module is responsible for applying tuning actions to a running application task. It utilizes solutions given by the Analyzer to modify the program during run-time. The Tuner dynamically changes the application execution manipulating in the application process memory via DynInst library. The Tuner module offers the following API:

```
struct Breakpoint
{
    char const * funcName; InstrPlace place;
};
void LoadLibrary (int processId, char const * libPath);


void SetVariableValue (int processId,
                       char const * varName,
                       char const * varValue,
                       Breakpoint * brkpt);

void ReplaceFunction (int processId,
                      char const * oldFunc,
                      char const * newFunc,
                      Breakpoint * brkpt);

void InsertFunctionCall (int processId,
                         char const * funcName,
                         int nAttrs,
                         Attribute  * attrs,
                         char const * destFunc,
                         InstrPlace   destPlace,
                         Breakpoint * brkpt);

void OneTimeFunctionCall (int processId,
                          char const * funcName,
                          int nAttrs,
                          Attribute  * attrs,
                          Breakpoint * brkpt);

void RemoveFunctionCall (int processId,
                         char const * funcName,
                         char const * callerFunc,
                         Breakpoint * brkpt);

void FunctionParamChange (int processId,
                          char const * funcName,
                          int paramIdx,
                          int newValue,
                          int * requiredOldValue,
                          Breakpoint * brkpt);
```

The API allows the Analyzer to perform a limited number of tuning actions:

- **LoadLibrary** – loads a specified shared library to a given application process. This enables the Analyzer to load any additional code required for the tuning.
- **SetVariableValue** – modifies a value of a specified variable in a given application process.
- **ReplaceFunction** – replaces all calls to function oldFunc with calls to function newFunc in a given application process.
- **InsertFunctionCall** – inserts a new function invocation code with a specified attributes at a given location in an application process.
- **OneTimeFunctionCall** – invokes one time a given function in a given application process.
- **RemoveFunctionCall** – removes all calls to a given function from the given caller function. For example this method can be used to remove all `flush()` function calls from a `debug()` function.
- **FunctionParamChange** – sets the value of an input parameter of a given function in a given application process. This parameter value is modified before the function body is invoked. There is also possible to change the parameter value under condition, namely if the parameter has a value equal to requiredOldValue, only then its value is changed to new one. If the requiredOldValue is zero, then the value of the parameter is changed unconditionally.

The Breakpoint parameter used in all tuning functions is used for synchronization purposes. The synchronization specifies when the tuning action can be invoked to ensure the correctness of an application. Currently, the tuner supports only the breakpoint-based synchronization. A breakpoint can be inserted into the application at the specific location (at the function entry or exit). When the execution reaches the breakpoint, the actual tuning action is performed and then the breakpoint is removed.

5.5.3. Dynamic monitoring library

This module (DMLib) provides the event tracing functionality and is implemented as a shared library. The library is loaded dynamically by the AC into the address space of each individual process. To load the monitoring run time library into the running process, the AC uses DynInst `BPatch_thread::loadLibrary()` method. The library contains functions that are responsible for registration of events with all required attributes and for delivering them for analysis.

The DMLib is developed in C/C++ language and offers the following API:

```
void DMLib_InitLogger (int processId,
                        char const * analyzerHost,
                        int analyzerPort,
                        long64_t clockDiff);
void DMLib_OpenEvent (int eventId, int nAttrs);
void DMLib_AddIntAttr (int value);
void DMLib_AddFloatAttr (float value);
void DMLib_AddDoubleAttr (double value);
void DMLib_AddCharAttr (char value);
void DMLib_AddStringAttr (char const * value);
void DMLib_CloseEvent ();
void DMLib_DoneLogger ();
```

The function `DMLib_InitLogger()` initializes the monitoring library by providing it the information about the monitored process, location of the analyzing server and the clock difference. During initialization the library establish the connection to the Analyzer process via TCP/IP protocol, registers itself so that later all generated events can be delivered to the Analyzer. The clock difference parameter represents the time difference between the local machine and the referenced one what allows for support of the global timestamp. The difference value is stored and later used to adjust the timestamp of generated events.

Function `DMLib_DoneLogger()` finalizes the work of the library. This function should be called as the last one. It releases all acquired resources (files, memory), flushes buffered events, notifies the Analyzer about the process termination and finally closes the connection with this module. When a process is about to terminate, dynamic library is automatically unloaded from the memory.

The library contains a set of functions for event registration. The registration consists of several steps that are performed in a sequence:

- **Open event** – `DMLib_OpenEvent()` – this action starts recording a new event. The following information is recorded:

   - event identifier – a unique number that indicates what happed, i.e. what function was called and at what place (entry or exit).
   - nAttrs – a number of event attributes to be recorded

- **Add attribute** – `DMLib_AddXXXAttr()` – this action should be repeated nAttrs times to record values of all necessary event attributes. This method enables the registration of a variable number of event attributes, each attribute having a distinct value type There is a separate function defined for each distinct value type. For example, if a monitored function has signature (int, char) and we want to record both attributes, two function calls should be performed from the library: first one to record a value of integer type parameter, second one to record a value of char type parameter. To perform these actions, the functions `Tracer_AddIntParam()` and `Tracer_AddCharParam()` should be invoked with the extracted values of the corresponding function parameters.
- **Close Event** – `DMLib_CloseEvent()` – this action indicates that the event recording is complete and the data can be delivered for the analysis.

To record a new event, the Monitor module builds and inserts into the application process a code snippet at a given location. When the snippet is invoked, it generates the event by capturing the required attributes and calling the DMLib functions to register the event. Figure 5.12 illustrates this process.

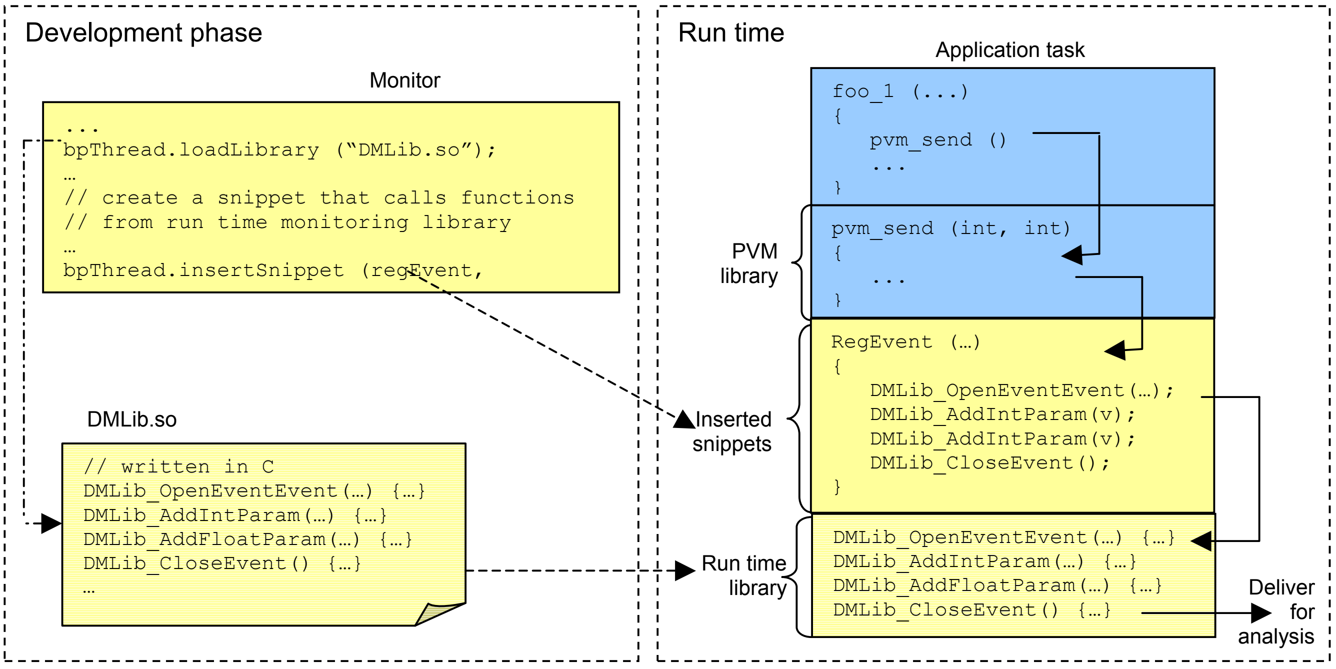


Fig. 5.12. Application instrumentation using the run time monitoring library loaded dynamically to the application.

The recorded events are delivered for analysis via simple event collection protocol based on TCP/IP. The protocol allows the DMLib to send messages that contain binary encoded sets of event records. Each event record is represented by the following data:

- **timestamp** – a number based on the system time `(gethrtime ())` that indicates when an event happened with high precision. As indicated previously the timestamp is adjusted to consider the clock differences to the reference machine.
- **event identifier**
- **number of attributes**
- **set of attribute values –** values of attributes recorded with the event

To minimize the network overhead, the DMLib implementation uses the event buffering mechanism. Instead of sending each individual event separately, there is an internal buffer used to group the events and send them in bigger messages. This allows for reducing the number of generated messages and limit the intrusion. One of the problems associated with buffering is related to delays that may occur before an individual event is physically sent. For example if a single event is registered and then during a period of time no other event is generated, the first event remains in the buffer and it is not delivered for the analysis. This problem is solved by using timers that automatically flush the buffer after a specified amount of time elapses.

5.5.4. Analyzer

This program carries out the performance analysis of the application, automatically detects existing performance problems "on the fly" and requests for appropriate changes to improve the application performance. The analysis is driven from the one side by application knowledge specified externally and from the other side by the online performance monitoring that is based on event tracing.

The operation cycle of the performance analysis process contains the following steps. When the Analyzer has been started, it starts the Application Controller. Once the AC is distributed all over the PVM virtual machine, the Analyzer receives from it information about the configuration of a virtual machine. During application execution, it is informed about all the changes in PVM virtual machine (e.g. a new task has been spawned, a task has terminated a host has been added or removed). The Analyzer contains a set of tunlets that in fact provide the performance analysis logic. Tunlets provide the Analyzer with an initial set of measure points. Next, the Analyzer forwards them to all ACs. This is done by sending the `AddEvent()` instrumentation requests. Then, the Analyzer requests the Master AC to start the application. When the application has started, the Analyzer enters in a bottleneck search phase. It continuously receives requested event records generated by different processes. When an event record comes, the Analyzer notifies corresponding tunlet and this tunlet in turn finds bottlenecks and determines their solutions. By examining the set of coming event records, the tunlet extracts measurements and then it evaluates a built-in performance model to determine the actual and optimal performance. If the tunlet detects a performance bottleneck, it decides if the actual performance can be improved in existing conditions. If it is the case, it then request the Analyzer to apply 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 the Tuner. For example, when the tunlet 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 (the function `FunctionParamChange()` must be invoked from the Tuner API).

Obviously, during the analysis, DMLibs are collecting and providing new data to the Analyzer. The tunlet may need more information about program execution to determine the causes of a particular problem or if a problem is already solved it may need no more a specific instrumentation. Therefore, the tunlet notifies about it the Analyzer module, that in turn is able to dynamically control the monitoring of the application by requesting more or less performance data to be collected. It can therefore request the AC, and hence the Monitor to change the instrumentation dynamically (functions `AddEvent()`, `RemoveEvent()` are invoked from the Monitor API).

The Analyzer program consists of a number of cooperating modules. Figure 5.13 presents its internal architecture. From functional point of view, the Analyzer is divided into two principal parts:

- **Dynamic Tuning API** – application programming interface for distributed performance monitoring and tuning of parallel program

- **Tunlets** – the modules that provide analysis logic and use the API to actually perform the dynamic tuning

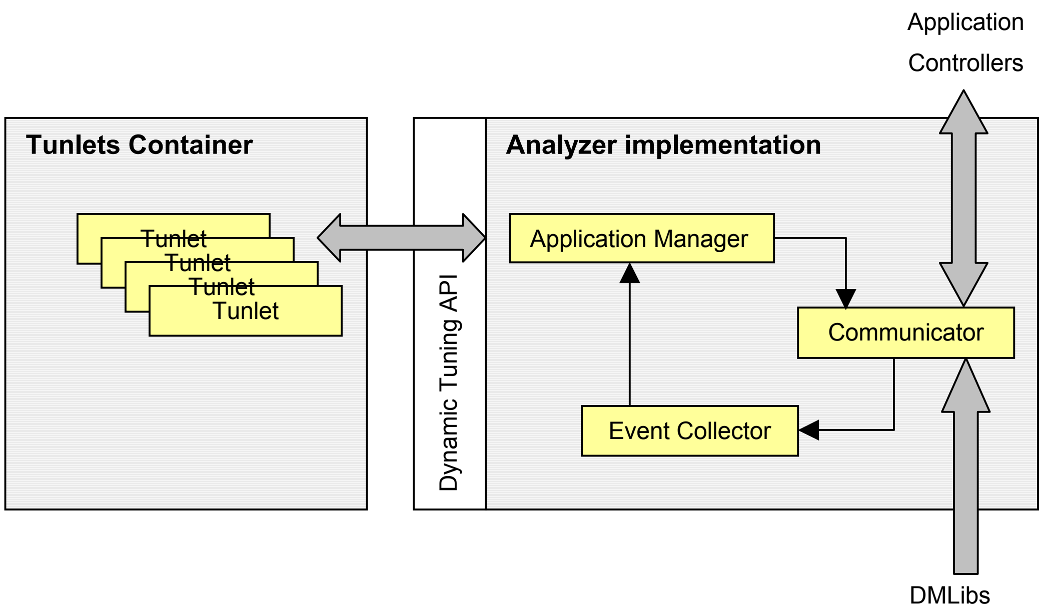


Fig. 5.13. Internal architecture of the Analyzer.

Dynamic Tuning API

This API encapsulates all low-level issues related to controlling the execution of the parallel application, its performance monitoring and tuning. It is implemented as a distributed asynchronous system where:

- the monitoring instrumentation and tuning service requests are delegated to distributed Application Controllers that in turn instrument and tune the application tasks
- the incoming events (event records sent by DMLibs and meta data sent by ACs) are collected and dispatched to registered event handlers.

In that sense, the architecture of this part of the Analyzer is similar to the DPCL library [Pas98].

The Analyzer implementation consists of three main modules: Communicator, Application Manager and Event Collector. The Communicator module provides connection with external world. The bi-directional communication is established with the Application Cotroller. The Analyzer is able to send monitoring and tuning requests to corresponding ACs. In turn, the Analyzer receives from ACs meta data about the application model (e.g. running tasks, hosts included into the PVM virtual machine). The Analyzer establishes unidirectional communication with DMLibs and it receives event records generated by each DMLib.

Since the Analyzer provides the global application analysis, it requires the actual information about the application distribution, monitoring events and tuning actions. We distinguished the Application Manager module that maintains the application model. It keeps the model up to date registering all changes. It actualizes the information about the running tasks, hosts where these tasks are running, as well as about the monitored events and tuning actions requested for each task.

When the event records arrive from distributed DMLibs, they must be preprocessed before they can be passed for analysis to corresponding tunlets. The module responsible for the preprocessing is called Event Collector. It stores a moving window of events incoming from different processes using a pool of buffers. The maximum size of this event window can be configured by the tunlets. Optionally, the Event Collector is able to reorder incoming events within the window and assure their global, casual order (e.g. receive cannot finish before send finishes).

The Dynamic Tuning API is provided as a collection of C++ classes as illustrated in Figure 5.14.

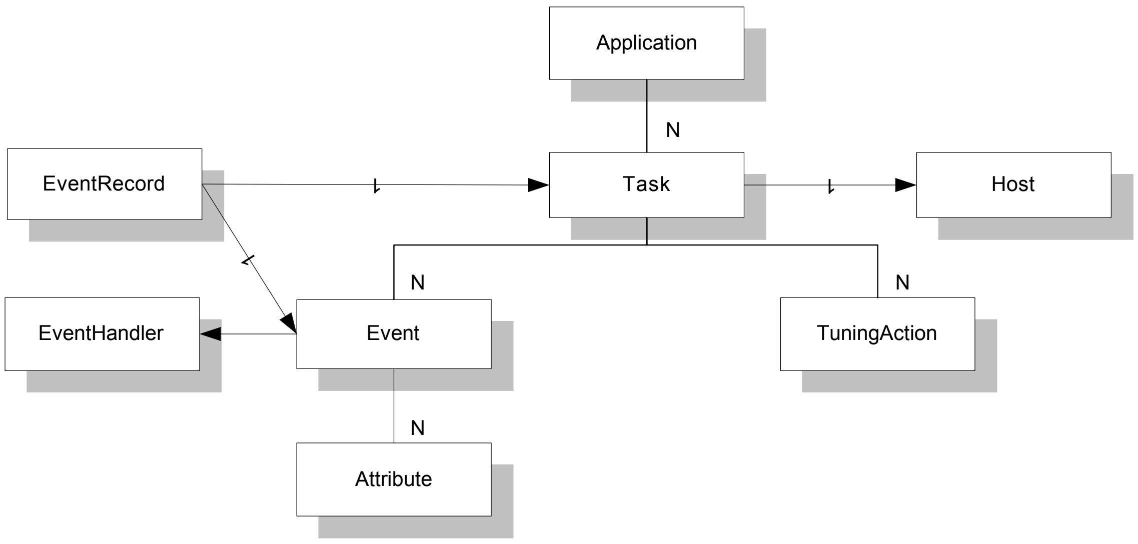


Figure 5.14. Dynamic Tuning API class diagram (simplified for clarity).

The analyzed application is represented by the Application object. The application consists of a number of Tasks. The collection of tasks inside the Application object is updated automatically to reflect the actual tasks of the running application. Each Task represents an individual application process (i.e. PVM task) and contains meta data (properties specific to that task (e.g. process identifier, host where the task is running). Each task may have a number of events to be monitored. A traced event is represented by the Event object. An event contains a set of Attribute objects that define what information should be recorded with the event. Each Event object is associated with an event handler that is called each time a record of the event occurrence is received by the Analyzer. In addition, the Task object contains a history of all tuning actions performed on that task.

The following sections describe in detail the classes and methods supported by the Dynamic Tuning API.

- **Application class**

  Properties
  - Name – name of the running program
  - NumActiveTasks – number of tasks actually running
  - Tasks – a collection of Task objects
  - Hosts – a collection of Host objects that form the virtual machine
  - MasterTask – references the master task of the application
  - Status – application status information
  - MonitoredEvents – collection of events being monitored in all the tasks

  Methods
  - Start – executes the application
  - AddEvent – adds a definition of new event to be traced in all running tasks of the application
  - RemoveEvent – removes previously added event from all running tasks
  - LoadLibrary – load a shared library to all running tasks
  - UnloadLibrary – removes a previously loaded shared library from all running tasks
  - SetVariableValue – modifies a value of a specified variable in a given set of tasks
  - ReplaceFunction – replaces all calls to a function with calls to another one in a given set of tasks

- InsertFunctionCall – inserts a new function invocation code at a given location in a given set of tasks
- InsertOneTimeFunctionCall – inserts a new function invocation code in a given set of tasks and calls it once
- FunctionParameterChange – sets the value of an input parameter of a given function in a given set of tasks
- RemoveFunctionCall – removes all calls to a given function from the given caller function in a given set of tasks

Callbacks

- SetTaskHandler – installs a callback function that is called when a new task is started or existing one is terminated
- SetHostHandler – installs a callback function that is called when a new host is added to the virtual machine or an existing one is removed

- **Task class**

  Properties

  - Id – globally unique task id
  - Name – process name
  - FilePath – file path of the task image
  - Host – reference to the host object this task is running on
  - IsRunning – indicates if the task is still running
  - Status – task status information
  - MonitoredEvents – collection of events being monitored in this tasks
  - TuningActions – a collection of tuning actions performed in this task

  Methods

  - AddEvent – adds a definition of new event to be traced in this task
  - RemoveEvent – removes previously added event from this task
  - LoadLibrary – load a shared library to this task
  - UnloadLibrary – removes a previously loaded shared library from this task
  - SetVariableValue – modifies a value of a specified variable in the running task
  - ReplaceFunction – replaces all calls to a function with calls to another one in this task
  - InsertFunctionCall – inserts a new function invocation code at a given location in this task

  - InsertOneTimeFunctionCall – inserts a new function invocation code in this task and invokes it once
  - FunctionParameterChange – sets the value of an input parameter of a given function in this task
  - RemoveFunctionCall – removes all calls to a given function from the given caller function in this task

  Callbacks

  - SetTaskExitHandler – installs a callback function that is called when this task terminates

- **Event class**

  Properties

  - Id – globally unique event id
  - FunctionName – name of the function this event is associated to
  - InstrPlace – function entry or exit
  - NumAttributes- number of event attributes
  - Attributes – a collection of attributes to be recorded with this event

  Callbacks

  - SetEventHandler – installs a callback function that is called each time a record of this event is delivered

- **Attribute class**

  Properties

  - Source – indicates source for attribute value (i.e. constant value, function parameter value, variable value, function return value)
  - ValueType – data type of the attribute value (i.e. integer, float, etc.)
  - SourceId – identifies the object to be used as a source (i.e. variable name, function name to be called, index of function parameter)

- **EventRecord class**

  Properties

  - EventId – globally unique event id
  - Event – references event object this record is associated to
  - Timestamp – indicates when the event happened
  - Task – references a task that generated this event
  - AttributeValues – a collection of recorded attribute values

- **EventHandler class**

  Methods

  - HandleEvent – called to handle an event record
- **TaskHandler class**

  Methods

  - TaskStarted – called when a new task is started
  - TaskTerminated - called when a task is terminated
- **HostHandler class**

  Methods

  - HostAdded – called when a new host is added to the virtual machine
  - HostRemoved – called when a host is removed from the virtual machine

Tunlets

The Analyzer provides a Tunlets Container (TC) module. This module is responsible for managing and running a set of tunlets simultaneously. Technically, each tunlet is a shared library that implements a particular tuning technique. Tunlets are assumed to be passive modules that drive the analysis by responding to a set of incoming events. The tunlet library is required to provide a very simple interface that consist of two standard entry points:

- Initialization – the tunlet library is initialized by the TC after it has been loaded. The TC initializes the tunlet by passing it the access to the Dynamic Tuning API. From this moment on, the tunlet is only invoked to handle events.
- Finalization – this functionality is called when a tunlet library is unloaded from memory

During initialization, the tunlet registers callback functions in order to receive events. This is performed by calling API functions. For example the tunlet may register handlers to receive notifications about changes in task configuration (Application::SetTaskHandler) or virtual machine configuration (Application::SetTaskHandler). It may also request to monitor the initial set of events (Task::AddEvent) for a particular task. When a record of a particular event arrives, it is delivered to the tunlet by calling a registered handler. The handler is then responsible to process the event and run analysis logic incrementally. When the analysis detects a performance problem it may use the API to change the requested instrumentation (e.g. Task::RemoveEvent) or request to perform a selected tuning action (e.g. Task::SetVariableValue). Finally, when the analysis is finished, tunlet is finalized and unloaded from memory. Figure 5.15 presents an example interaction diagram that shows the sequence of calls between container, tunlet and Dynamic Tuning API.

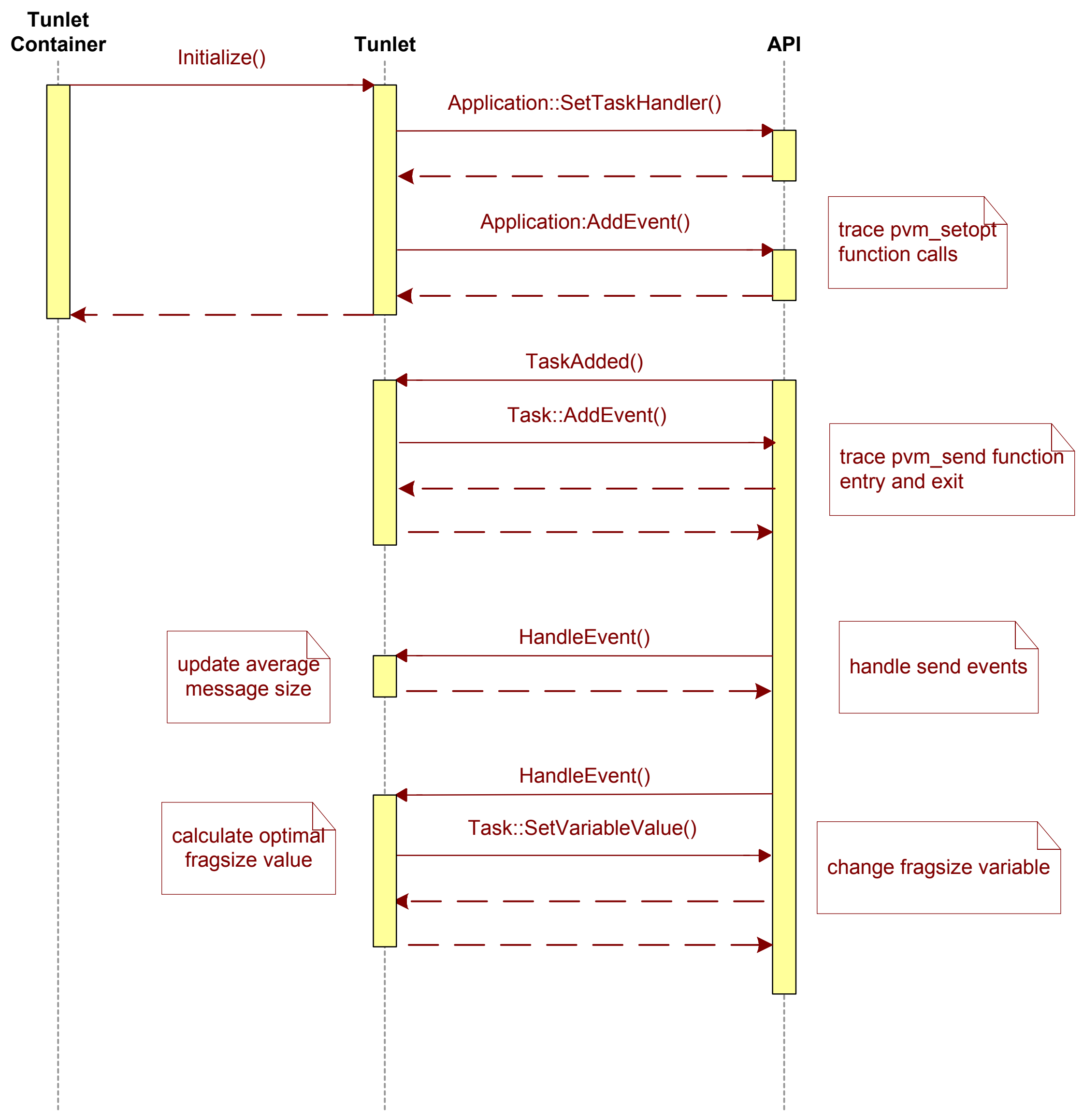


Fig. 5.15. Sample interaction diagram between Analyzer modules.

5.6. Restrictions and limitations

There are several constraints not contemplated in MATE that we consider interesting for future investigations. First of all, the performance measurement is based on event tracing. This enables the Analyzer to have a very insight view on the behavior of the application, however in some circumstances the associated overhead may not be unacceptable. For example the cost of collecting individual send/receive calls may introduce too high network intrusion for communication intensive applications. In such conditions, it would be more reasonable to calculate communication statistics such as average message size, total communication time, etc. inside the application task rather than collect the events. The dynamic profiling technique based on insertable instrumentation would be in that case an interesting, but complementary solution.

Another limitation results from a fixed set of tuning actions in the Dynamic Tuning API. Although these actions cover a range of possible application changes, there are situations when more flexibility in generating inserted code is required (e.g. conditional tuning actions). In that sense, the API could be extended to support dynamic definition of tuning code. This could be achieved by defining a rich API (like DPCL) or defining a scripting language that allows one to express a generic instrumentation code.

5.7. Conclusions

One of the principal goals of this work was to create a dynamic tuning environment that is able to automatically tune the application performance during run time. We devoted a big attention to its creation and our development concluded in the working environment called MATE. In general, it includes the monitoring, analysis and tuning of the application on the fly without stopping, recompiling or rerunning it. MATE is suitable for the applications that do not have a stable behavior and/or change from run to run according for example to the input data or to the environment. We determined the set of requirements that such a tool should meet and then designed and implemented a software taking into consideration all of them. MATE provides a set of facilities to support dynamic monitoring, detection of performance bottlenecks and automatic changes of the running application. Moreover, our environment provides the programming models that allows for implementation of new tuning techniques that solve concrete performance problems. Currently, our environment can be treated as the prototype for complete future implementation and there are still many aspects that remain for considerations and improvements.

Our approach is based on the closed steering loop. Therefore, MATE is in some sense similar to e.g. Falcon or SCIRun as we modify the application behavior at run-time. However, MATE is not a Problem Solving Environment, because we focus on the performance optimizations of the application. Moreover, the steering is automatic, there is no interaction with a user. MATE detects problems in applications and tunes them on-the-fly without user intervention.

Our tool is related to Active Harmony and Autopilot. Although, MATE presents similar tuning approach to the Harmony and Autopilot, it differs in many assumptions and details. The MATE environment provides techniques for cooperative usage where application must be prepared for the changes. The same situation appears when using Autopilot or Active Harmony. However, MATE also tries to go more into the automatic black box direction where all the tuning phases can be done automatically without user intervention. Moreover, there are other evident differences as: performance analysis models, instrumentation, and development which we have mentioned in the paragraph describing Autopilot and Active Harmony projects.

Chapter 6

Tuning techniques

This chapter focuses on the presentation of the catalog of the tuning techniques. First, we introduce the organization of the catalog and show the scheme of the tuning technique description. Next, we present and justify the software and hardware components that were used for practical experiments conducted with the techniques. In continuation, we present a set of tuning techniques. Each technique is described in a systematic way giving us a global view of the performance problem it addresses, its general applicability, solution it applies, experimental results and benefits it gives. Final section summarizes and concludes the catalog pointing directions for future work.

6.1. Introduction

As we have mentioned in Chapter 5, we provide our dynamic tuning environment with an application knowledge that represents specific, determined information about performance problems that can occur during application execution and solutions to these problems. All required information related to one particular problem we called a **tuning technique**. Each tuning technique describes a complete performance optimization scenario, namely:

- It specifies a potential performance problem of a distributed parallel application
- It determines what should be measured to detect the problem (set of measure points)
- Given the measurements, it determines how to detect the problem (performance model)
- It provides a solution on how to overcome the problem (tuning point/action and synchronization).

A set of tuning techniques forms a **catalog**. We have organized the catalog in accordance with the tuning layer at which tuning occurs. Each particular tuning technique is implemented in the MATE environment as a **tunlet**. The tunlet contains specific code related to one concrete bottleneck that can occur in the application and its solution.

Currently, we are focusing on investigating tuning techniques separately. During the application execution, MATE attempts to apply all of the optimization scenarios, but each one individually. MATE loads available tunlets and each of the incorporated tunlets carries out the application optimization. When the MATE environment starts the application execution, each tunlet performs the analysis of a specific problem. It requests to monitor the appropriate events, receives event records, and analyses them detecting only the addressed problem. When a problem is found and its solution is determined, the tunlet requests the tuning actions.

Our goal was to identify and investigate different tuning techniques. Therefore, we focused on the effects of individual techniques. However, we do not take into consideration the overall performance of the application. For example, if the communication time is very low, it may be advisable not to use any of the communication tuning techniques. Moreover, in certain conditions it may be necessary to consider dependencies between different performance problems and associated tuning techniques. These issues are left for further investigations.

In the following sections we present a catalog of individual tuning techniques on which we mainly focused our work and that we studied within the MATE environment.

6.1.1. Catalog organization

We classified the catalog of tuning techniques into two main parts concerning two different approaches to the dynamic tuning described in Chapter 4: automatic and cooperative. Both of them are consequently divided into 3 subparts, that present different tuning layers, namely operating system, library and application. For each subpart we can distinguish specific tuning techniques.

We present the following catalog of tuning techniques:

1. **Automatic approach**
    - **Operating system level** – the described tuning techniques focus on the usage of the operating system functionality
        - Message aggregation
        - TCP/IP buffers
    - **Standard library level** – the described tuning technique focuses on the usage of the C library functionality
        - Memory allocation

- **Custom library level** – the described tuning techniques focus on the usage of the PVM library functionality
    - PVM communication mode
    - PVM encoding mode
    - PVM message fragment size

2. **Cooperative approach**
    - **Application level** – the described tuning techniques focus on the tuning of the application-specific problems
        - Workload balancing (factoring)
        - Number of workers

6.1.2. Technique description

The description that we use to present a tuning technique is based on the description of a design pattern written in the Object Oriented Design Patterns book [Gam95]. A design pattern contains the following sections that describe it: intent, motivation, applicability, consequences, implementation, example/design, known uses, related patterns. We found the organization of design patterns very systematic, proper, clear and useful for a good problem presentation. Therefore, we intended to follow this model and hence each description of a tuning technique has a set of the following sections:

- **Intent** – this is a short description (1 sentence) of the performance problem addressed by the tuning technique.
- **Motivation** – this explains a typical, representative performance problem that a tuning technique deals with. It discusses why the presented problem exists in a parallel and distributed application and why such a tuning technique is needed.
- **Applicability and conditions** – this section describes when a tuning technique can be applied and a list of conditions that must be satisfied for the tuning technique to be useable. The conditions express criteria that the tuning technique must consider in order to improve the performance of an application.
- **Solution** – this section gives a solution to the presented problem. It explains how to detect the problem in a running application and what should be changed to improve the performance.
- **Implementation** – it presents details of the tuning technique implementation (tunlet implementation). This section details in what way the tunlet cooperates with the MATE

environment, what measure points it provides and how it analyses incoming event records and detects the problem. Finally, the tuning action is described together with the points on which it must be invoked as well as the synchronization mechanism.

- **Experiment** – this section presents the results of practical experiments with applications and the performance dynamic tuning that we were able to conduct. Each presented experiment contains the description of the tested application, as well as the execution scenario. Finally, this section shows and discusses measurements and results obtained from the conducted experiments.
- **Conclusions** – it concludes the presented tuning technique.

6.1.3. Environment description

All our experiments were conducted in an environment consisting of a cluster of workstations connected by LAN network. We used the workstations available in our laboratory. The environment contained 5 machines connected by Ethernet 10/100 Mbps network. All the machines except Aows10 were equipped with 100Mb/sec Fast Ethernet adapter and they could take advantage of the faster network. Detailed configuration of the environment is shown in Table 6.1.

No.	Machine name	Type of CPU	Memory size	Operating system	Relative speed	Comments
1.	aows10.uab.es	Sun UltraSPARC I, 167 MHz	128 MB	Sun Solaris 2.6	1.00	Main NFS server, 10Mb network adapter
2.	aows1.uab.es	Sun UltraSPARC II, 440 MHz	128 MB	Sun Solaris 2.8	2.76	100Mb network adapter
3.	aows6.uab.es	Sun UltraSPARC II, 440 MHz	128 MB	Sun Solaris 2.8	2.76	100Mb network adapter
4.	aows7.uab.es	Sun UltraSPARC II, 440 MHz	128 MB	Sun Solaris 2.7	2.77	100Mb network adapter
5.	aows8.uab.es	Sun UltraSPARC II, 440 MHz	128 MB	Sun Solaris 2.7	2.79	100Mb network adapter

Table 6.1. Configuration of the experimental environment.

Since our environment comprised machines with different hardware configurations, it became necessary to consider their hardware capacities. For obvious reasons, the execution time of a program running on a workstation depends directly on the hardware capacities of the machine. This time depends not only on processor speed, but also on memory size and access time, cache memory parameters, hard disk parameters, to mention the most important. While these parameters may give an indication of the machine capabilities, they cannot be simply applied to calculate the machine performance, hence another method is necessary. Reliable and accurate performance metrics can be obtained by running benchmark programs. Benchmarking measures the time needed to execute a selected computing task on many machines hence it allows for making performance comparisons. For the purpose of some of our experiments (e.g. workload balancing), we estimated the relative workstation speed using Whetstone benchmark [Cur76]. The results are presented in the column "Relative speed" in Table 6.1. The primary goal of this benchmark is to provide a performance measure of both floating point and integer arithmetic. Therefore, it is well suitable for scientific applications and not for general evaluation of efficiency. The relative speeds are calculated as the ratio between the MWIPS value (Millions of Whetstone Instructions Per Second) measured by the Whetstone benchmark on a given workstation and the MWIPS value measured by the same benchmark on a reference machine. We used slowest workstation, aows10, as the reference machine.

Finally, it must be pointed out that neither the machines comprising the cluster nor interconnection network was completely dedicated to purpose of the experiments. During all experiments, the aows10 machine was running mail-server application and Aows1 was running web-server application. Moreover, the performance of machines and the LAN network could have been affected by other users or programs. We tried to performed experiments under very low or no external load conditions avoiding interruptions, i.e. most of the experiments were conducted during night-hours when there is the lowest probability of the machine usage by other people. Therefore, the measurements we show do not have exact precision. To obtain more precise results, each experiment was repeated a number of times and the average of the wall clock execution time was calculated.

6.2. Message aggregation

This tuning technique intents to minimize communication overhead by transparently grouping set of small messages into large ones.

6.2.1. Motivation

Parallel applications usually generate a large amount of messages. If the problem decomposition is fine grained, the size of the transmitted messages is usually small. The overhead for sending these messages over an interconnection network (LAN) can dramatically limit the application speedup because of network latencies. Parallel applications such as event-based simulations or even parallel matrix multiplication programs are an area where all possible optimizations on communication are generally welcome because these applications typically have a high communication to computation ratio.

In this case, message aggregation technique can transparently increase the granularity of the transmitted messages and reduce the communication overhead. The message aggregation is based on the idea of grouping set of small messages for the same destination into a single larger one [Pha99]. The rational behind message aggregation is that it is cheaper to send an *M* bytes message than to send *n* times an *m* bytes message with $n * m = M$. This is true when a network latency is non-zero value and n > 1.

$n * m = M$

$T_{comm1} = T_{startup} + M * T_{word}$

$T_{comm2} = n * (T_{startup} + m * T_{word})$

$\mathbf{T_{comm1} < T_{comm2}}$ **?**

$T_{startup} + M * T_{word} < n * (T_{startup} + m * T_{word})$

$T_{startup} + n * m * T_{word} < n * (T_{startup} + m * T_{word})$

$T_{startup} * (n-1) > 0$

True if $\mathbf{n > 1}$ **and** $\mathbf{T_{startup} > 0}$

There is a trade-off on how long to keep aggregating before sending the message to the receiver. It the time is too long, it may produce useless waiting as a side-effect at the receiver side. It is certainly not desirable to wait too long, but at the other side too short delays may not benefit from aggregation.

6.2.2. Applicability and conditions

This technique is best suited for applications that use TCP/IP protocol, for example PVM-based parallel applications or any other communication software that makes use of sockets. The message aggregation works best when an application is executed in networks with considerable latencies (i.e. LAN or WAN networks) and sends a large number of small messages. This optimization can be performed at the low level of the communication software (i.e. OS sockets) or in the application at the higher level (e.g. PVM library). In our work we focus on socket level optimization.

6.2.3. Solution

To transparently aggregate a set of messages it is necessary to introduce an additional layer of code that provides the message buffering mechanism. The mechanism is based on a buffer with a sufficient size to store a set of small messages and an aggregation algorithm. The following pseudo-code explains the functioning of the aggregation mechanism:

```
      if (!aggregation)
            write message with std write call
            return
      if (curAggrBufIdx + msgSize < AGGR_BUF_SIZE) // message fits
aggrBuf
            copy message to aggrBuf
            set timestamp and flushing alarm
            curAggrBufIdx += msgSize;
            return
      else  // message is larger than aggrBuf
            if (curAggrBufIdx > 0) // there is data in aggrBuf
                  write data from aggrBuf with std write call // flush
      write message with std write call
      reset flushing alarm
      curAggrBufIdx = 0;
```

When a message is sent by the application, it is first intercepted by the aggregation mechanism. Next, in function of the message size and free space in the buffer, the algorithm decides if the message can be stored in the buffer or it is necessary to flush buffer content by sending one large message. Additionally, to avoid the situation when a message is stored in a buffer and the receiver waits too long for its reception, it is necessary to use time based automatic flushing mechanism. This mechanism ensures that all messages stored in a buffer are transmitted after a specified amount of time.

From the performance point of view, the message aggregation mechanism produces a *hit* when a small message is written to the buffer instead of being sent (i.e. it is aggregated) or *miss* when a message does not fit into the buffer. In the first case, the mechanism gives the performance gain, because the total number of transmitted messages gets reduced as well as the number of system calls. In the case of the miss, the performance is worsen in comparison to scenario without aggregation, because there is a cost resulting from flushing the content of the buffer (i.e. previous messages), sending the actual message and the overhead introduced by extra layer of code. Therefore it is necessary to analyze the sequences of outgoing messages and estimate the possible hit/miss ratio before taking decision on applying the aggregation mechanism.

To analyze whether the message aggregation should be used, it is necessary to perform a number of measurements. The goal is to check if the application is sending a big number of small messages consecutively. This can be achieved by collecting statistics of the number, sizes, and differences between sizes of consecutive messages being sent during a specified time window. These statistics must be collected for each individual connection (i.e. socket) separately as the aggregation is performed per connection. Basically, there are two conditions that must be met to consider this technique profitable:

- The number of small messages should be significant, i.e. the percent of messages with size smaller than `MinMessageSize` exceeds a threshold `MinPercentSmallMessages`.
- The estimated hist/miss ratio, i.e. probability of occurrence of sets of consecutive small messages (hits) exceed `MinHits`.

When these conditions are met for a given time window, there is a certain probability that if the application does not change radically its behavior, the application of the message aggregation mechanism can be beneficial for the performance. In that case, the tuning actions are triggered. The tuning consists of replacing operating system function calls that transmit data, in particular `write()` function, with their optimized version with aggregation mechanism, i.e. `aggr_write()` function.

The advantage of carrying out the message aggregation at the level of system calls is the transparency on the receiver side. It is not necessary to introduce any changes on the receiver, because the mechanism does not affect the data being transmitted (i.e. receiver gets exactly the same data in the same order). The only difference is on the sender side, where small messages are copied to the intermediate buffer, the number of system calls gets reduced, but each call carries more data to be written to the underlying TCP buffer. This independence enables the transparent usage of the aggregation mechanism with higher level communication software such as PVM, MPI and other libraries.

6.2.4. Implementation

To monitor if the application is sending a big number of small messages consecutively, the message aggregation tunlet must collect the statistics of the following events:

- `socket()` – to track file descriptors associated with network sockets. For each opened socket the tunlet collects the statistics for outgoing messages.
- `write(int fd, void * buf, int bufsize)` system function call – to calculate the histogram of number of writes for each data size being written, and also analyze sequences of consecutive message sizes in order to estimate the hit/miss ratio. These statistics must be collected separately for each open socket.
- `close()` – to get notified when the socket is closed

The processing of these measurements allows the tunlet to calculate the following metrics for a given socket and time-window:

- Histogram of number of writes for each data size written – this permits to estimate if the number of small messages can be considered significant
- Number of hits and misses – this enables the profitability analysis, i.e. estimation whether the technique can optimize the performance in a given conditions

When the required conditions are met, the tunlet can then invoke the appropriate tuning actions. Because of the complexity of the aggregation mechanism, its implementation is provided as an external shared library that is loaded into the application process on request. The library provides an API with the following set of functions:

- `void aggr_init()` – this function initializes the aggregation mechanism (e.g. variables, alarms, descriptors).
- `void aggr_open(int fd)` – this function activates the aggregation mechanism for a given socket.
- `void aggr_write(int fd, void const * buf, size_t bufSize)` – this function is used to replace the standard write function and contains the aggregation mechanism.
- `void aggr_flush()` – this function replaces the standard `flush()` function. When an application requests the flushing, it is first necessary to flush the content of the aggregation buffer.

- `void aggr_close(int fd)` – this function disables the aggregation mechanism for a given socket. From this moment on, all subsequent calls to `aggr_write()` will delegate the work directly to the `write()` function.
- `void aggr_set_flush_timeout(struct timeval * timeout)` – this function simple sets the maximum value of the timeout that may pass to flush the data that remains in the aggregation buffer.

The library implementation holds an array of data structures for each possible file descriptor. Those descriptors that are opened for aggregation with the `aggr_open()` call, the data structure contains the aggregation buffer and set of auxiliary variables such as current buffer index, size, timer data and so on. When the `aggr_write()` function is called, the code first checks if the file descriptor has the aggregation activated. If it is not the case, the call is immediately delegated to standard `write()` function. If the aggregation is active, first the message size is checked if it fits into the buffer. If this is true the message is written into the buffer and the call terminates with a hit. Otherwise, the current buffer content must be flushed with a single `write()` call and then the current message is also transmitted immediately. In addition to normal flow, the library implementation uses alarm-based timers in order to proceed with auto-flush procedure whenever the oldest message remains in the buffer for more than `MaxMessageAge` milliseconds. This prevents the aggregation mechanism from introducing long delays in message delivery.

The activation of the message aggregation mechanism consists of the following phases:

- The message aggregation library is loaded into the application process. This is performed by means of `LoadLibrary()` call from the Tuning API.
- Set of system calls (i.e. write, flush, close) gets replaced with their aggregating versions. This is performed by means of `ReplaceFunction()` call from the Tuning API.
- Then individually for each connection that should use the aggregation it is necessary to invoke the `aggr_open()` function call to enable the mechanism for a given socket. This is performed by means of `InsertOneTimeFunctionCall()` of the Tuning API.

6.2.5. Experiments

In order to **evaluate the effectiveness of message aggregation in comparison with standard message transmission in the TCP/IP protocol**, we have developed a simple, synthetic, socket-based master/slave program that transmits a series of consecutive small messages. The master task sends a determined amount of work divided into a number of messages and waits for confirmation from the slave. The slave receives the messages, and immediately sends the confirmation to the master. There is no computational operations, the application is strictly based on communication. We executed the program using 4KB work size. The same work was transmitted using various number of small messages ranging from 1024 messages with 4 byte size (i.e. one integer) each to 2 messages with 2048 bytes size each. We have performed the experiment in three different configurations:

- **Standard write** – without message aggregation mechanism
- **Aggregate write A** – using first aggregated write implementation where the buffer is not sent till all bytes of the message are acknowledged to be sent. Aggregation buffer size was 256B and the flushing timeout was set to 500miliseconds.
- **Aggregate write B** – using another aggregated write implementation in which additionally to the mode A, aggregation mechanism is also used before the message is write to the network stream. Aggregation buffer size was set to 256B and the flushing timeout to 500miliseconds.

All executions were conducted in homogeneous and dedicated scenario (no external load). In this scenario, we have used two homogeneous machines (aows6, aows7) connected by 100Mb/sec network. During execution of the experiments, all workstations were idle.

Measurements and results

Figure 6.1 presents the comparison of communication performance between different data transmission configurations: standard write, aggregate write mode A and mode B in the master/slave application presented above (note that Y scale is logarithmic). The detailed measurements are listed in Table 6.2.

We can observe that in function of work decomposition (message size and number of messages) the total time needed to transmit the data differs significantly (from –113% up to +91%).

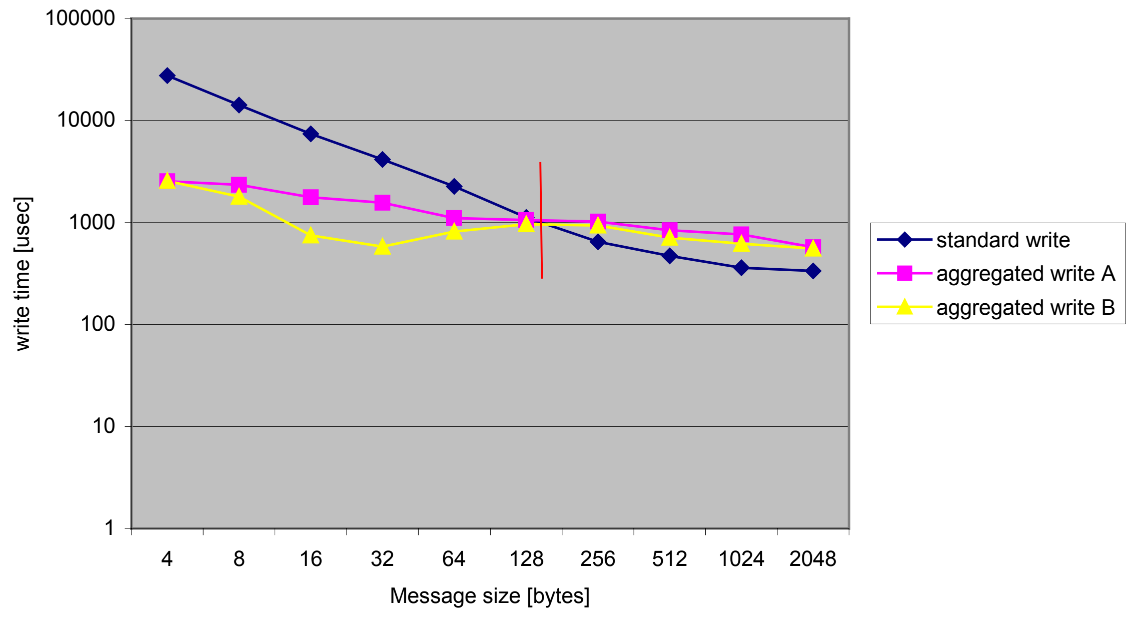


Fig. 6.1. The comparison of write performance in different configurations.

MsgSize [B]	Standard write time [usec]	Aggregated write A [usec]	Benefit from A	Aggregated write B [usec]	Benefit from B
4	27697	2543	91%	2565	91%
8	14167	2347	83%	1792	87%
16	7428	1771	76%	746	90%
32	4159	1560	62%	582	86%
64	2256	1105	51%	814	64%
128	1122	1061	5%	961	14%
256	645	1024	-59%	933	-45%
512	470	838	-78%	709	-51%
1024	359	766	-113%	619	-72%
2048	336	571	-70%	557	-66%

Table 6.2. The detailed measurements of write performance in 3 different configurations: standard write and two modes of the aggregated write.

For messages smaller than a cut-point that we estimated to be 156 bytes (see vertical line on the graph), the performance of the both configurations with message aggregation overcomes the standard `write()` system call. This can be explained by the sum of network latencies (i.e. $T_{startup}$ time) associated with each message being sent. Moreover, each `write()` system call requires the context switch from user to kernel mode and this also has its associated cost. When the message size grows and number of transmitted messages gets reduced, the sum of latencies and `write()` calls is lower and it is getting closer the aggregation cost. When the message size exceeds 156 bytes, the aggregation performance is worse than the standard `write()` performance. The costs of additional code layer and extra data copy start to overcome the benefits and the technique is no longer profitable. We may conclude that there is a limitation on the size of aggregated messages for the technique to be profitable.

For small messages we can see a difference between mode A and B of the aggregation implementation. This might be caused by the additional aggregation introduced before the message is written to the network stream.

6.2.6. Conclusions

This tuning technique can improve the communication performance for applications that frequently exchange sets of consecutive and small messages. The tuning cost is not very low, because it is necessary to load an additional shared library and then accept the constant run-time overhead resulting from additional layer of code that encapsulates some of the system calls (i.e. `write()` function). However, in networks that are characterized with not ignorable latencies and applications that send very small messages (e.g. no bigger than hundreds of bytes), the technique can bring noticeable time savings.

6.3. TCP/IP buffers

This technique intents to maximize the network transmission performance across high-performance networks using TCP/IP-based protocol. This is achieved by tuning the TCP socket buffers to an optimal value.

6.3.1. Motivation

In order to take full advantage of high speed networks and maximize communication performance, it is necessary to pay attention to some of the configuration and communication tuning issues. In particular, it is necessary to tune TCP/IP protocol to achieve high data transmission rate over the fast networks. Some of the issues discussed below arise because of the fact that the modern networks have been improved beyond what they were when the TCP/IP protocols were initially designed. Although there are some high performance extensions that have been proposed and implemented in the TCP/IP protocol, these options are sometimes not enabled by default and require the programmers to take care of them manually.

TCP/IP is a reliable and window-based protocol. Under ideal conditions, best possible network performance is achieved when the data pipe between the sender and the receiver is kept full. The amount of data that can be transferred in the network, sometimes called **Bandwidth-Delay-Product** (BDP for short), is simply the product of the bottleneck link bandwidth and the Round Trip Time (RTT). In a reliable protocol such as TCP/IP, the importance of BDP described above is significant as this represents the amount of buffering that will be required in the end hosts (sender and receiver).

The largest buffer size in the original TCP/IP (without the high performance options) is limited to 64KB. If the BDP is small either because the link is slow or because the RTT is small (in a LAN, for example), the default configuration is usually adequate. But for a paths that have a large BDP (i.e. "Long Fat Networks"), and hence require large buffers, it is necessary to have enabled the high performance options discussed below.

For TCP/IP protocol, the window size option is by far the most important parameter to adjust for achieving maximum bandwidth across high-performance networks. Properly setting the TCP window size can often more than double the achieved bandwidth. With TCP, each segment header contains a field called "advertised window" specifying how many additional bytes of data the receiver is prepared to accept. The "advertised window" may be interpreted as specifying the receiver's current available buffer size. An important fact about TCP is that the sender is not allowed to send more bytes than the advertised window. This is TCP's flow control mechanism. To maximize the performance, the sender should set its send buffer size and the receiver should set its receive buffer size to no less than the capacity of the TCP pipe. As stated before, theoretically this number should be equivalent the product of bandwidth and round-trip-time (BDP).

The default values of TCP/IP buffer size (i.e. socket buffer size) differ widely between implementations. Older Berkeley-derived implementations would default the TCP send and receive buffers to 4KB, but newer systems use larger values (up to 64KB). The new TCP extensions support values up to 1 MB or more.

In our work, we intent to set the TCP socket buffers to an optimal value by detecting the bandwidth-delay product at connection setup time. The objective is to maximize the data transfer rate, even if the default socket buffer size is small. For those connections with a low bandwidth-delay product, we assume to leave the socket buffer size small in order to conserve memory usage.

6.3.2. Applicability and conditions

This technique is best suited for applications that use TCP/IP protocol to realize bulk transfers between two end-points in fast networks with large BDP. For example parallel applications that transfer large volumes of data between tasks running in GRID environments are good candidates for this optimization. The same applies for traditional Internet data transfer applications such as FTP clients and servers.

It must be pointed out that the TCP buffer tuning is only possible at TCP connection setup time and remains the same within a single TCP session. The mechanism to be effective needs that both sender and receiver use the same or at lease similar buffer size value (send buffer and receive buffer respectively). Therefore the tuning is more complex because it requires to perform modifications on both sides at the same time.

Another limitation of socket buffer tuning is the amount of memory that may be used by an application for buffering network data. Some older operating systems limit this value to 64KB. In a newer UNIX implementation that supports RFC1323 "Large Windows" extension, a maximum value for the socket buffer size varies between 128 KB and 1 MB. For example the Sun Solaris 2.8 that we used for experiments permits 1MB value.

6.3.3. Solution

Theoretically the TCP window size should be set to the bandwidth delay product, which computes the volume of data that can be in the network between two machines. The bandwidth delay product is:

```
BDP = bottleneck bandwidth * round-trip time
```

To compute the BDP it is necessary to find out the speed of the slowest link in the path between two communicating nodes and the round trip time. The bandwith of a link is typically expressed in Mbit/s. The round-trip time (RTT) for a WAN link is typically between 10 and 100 milliseconds.

Since we set TCP buffer size to the same value as the available bandwidth-delay product, the problem is essentially to estimate the available bandwidth and RTT with minimum cost of time and traffic and at least a coarse-grained accuracy. To achieve this we use the sequence of the steps as described in [L39]:

- We send out a series of fixed-length ICMP_ECHO packets as fast as possible. Each packet is time-stamped and it has an associated unique sequence number.
- Measure the RTT by comparing the arrival time of the echoed packets with the time-stamps contained in the packets.
- Record those packets that arrive in-order (consecutive sequence numbers) and the inter-arrival times. The packet length divided by the inter-arrival time is the assumed available bandwidth.
- Determine the bandwidth-delay product (BDP) by multiplying the bandwidth numbers by their corresponding RTTs
- Finally multiply the median of the products by a constant factor (1.0 in our implementation) and return it as the result of our measurements.

For example if we send a series of 256 bytes long packets and they are echoed back within 10 milliseconds, the inter-arrival time is about 0.05 milliseconds, we obtain:

```
RTT = 10 milliseconds

Bandwidth = 256 bytes / 0.05 milliseconds
          = 256 bytes * 1000 / 0.05 seconds = ~39Mbps
(e.g. this can be 100 Mbit/sec Ethernet link under load)

BDP = RTT * Bandwidth * ConstFactor
    = 10 milliseconds * 39Mbps * 1.0 = ~50KB
```

Having the solution to estimate the BDP, we are now able to define a tuning procedure. Because it is only possible to change the socket buffer sizes before the connection gets established, the applicability of this technique is limited to the setup time of a given connection. Therefore, it is necessary to intercept the `connect()` procedure at the sender side and the `accept()` procedure at the receiver side. In both cases, before the connection is requested by the sender or accepted by the receiver, it is necessary to invoke BDP estimation procedure and use the resulting value to setup the size of socket buffers. To set these buffers, we invoke `setsockopt()` system call using the `SO_SNDBUF` for the sender and `SO_RCVBUF` socket option for the receiver.

It can be observed that this solution differs from other tuning techniques, because the tuning decision must be taken at the application startup time and before any run-time knowledge is available at least for the first connection. Although this optimization is performed automatically for any socket-based application, it is advisable that the programmer decides to use the technique or not for his/her application. This results from the limitation of TCP/IP socket API that does not allow to change the buffer sizes for an established connection.

6.3.4. Implementation

The implementation of this tuning technique is divided into the following components:

- Automatic BDP estimation library – shared library module that is loaded into the application during its run-time and provides the functionality that enables the estimation of the bandwidth-delay product for an individual connection.
- Tunlet – executed inside the MATE Analyzer. It uses Tuning API to load the library to the application processes (both senders and receivers) and to instrument system calls, i.e. `connect()` and `accept()`by adding invocations to BDP estimation code.

In particular, the BDP estimation library provides the following interface:

- `int EstimateBDP(char const * remoteHostName)` – returns estimated bandwidth-delay product in bytes for a connection with a given remote host using ICMP based method as described in the Solution section.
- `void AutoSetupBuffers(int socket)` – automatically sets up the send and receive buffers of a given socket by applying the estimated BDP value. This function internally invokes the `bdp=EstimateBDP()` function and then calls `setsockopt(socket, SOL_SOCKET, SO_SNDBUF, &bdp, len)` and `setsockopt(socket, SOL_SOCKET, SO_RCVBUF, &bdp, len)` to setup the buffers.

To use the BDP estimation, the tunlet requests the MATE Application Controllers to insert the `AutoSetupBuffers(socket)` function call into the entry of the `connect()` function and at exit of the `accept()` function. Because the TCP window size is implemented by send and receive buffers on each end of the connection, this operation must be invoked for both ends of the connection.

6.3.5. Conclusions

This technique tunes the TCP socket send and receive buffers in order to maximize the network transmission rate. It is best suited for applications that realize massive data transfers between two end-points. The automatic buffer tuning may be very beneficial, [L39] reports performance enhancement up to 500% for FTP transmissions (i.e. transfer rate improved from 60Kbps up to 420Kbps). However, the application of this technique is limited, because the buffers can only be tuned before the connection is established. It is not possible to adapt their size during run-time by analyzing the transmitted data. Therefore it is advisable that the programmer decides to apply this technique or not for a particular application.

6.4. Memory allocation

This tuning technique intents to improve the performance by optimizing the memory allocations for applications that use big number of small objects.

6.4.1. Motivation

Machines with large amounts of memory and disks are very common. However, current general-purpose memory allocators do not provide sufficient speed or flexibility for modern high-performance applications. To achieve high performance, programmers often write custom memory allocators from scratch. Many general-purpose memory allocators implemented in C and C++ provide good performance for a wide range of applications, but using specialized memory allocators that take advantage of application-specific behavior can dramatically improve application performance [Ale01].

Typically, the applications use general-purpose memory allocators (i.e. `malloc()`, `free()`). Standard allocators are known to be inefficient in particular cases, for example when using a large number of small objects. C programs usually allocate medium- to large-sized structures (hundreds to thousands of bytes) and for such behavior `malloc()` and `free()` are optimized. However, in many applications (especially written in C++), there is a tendency to create quite large number of small objects (tens to hundreds of bytes). This can result in the application performance problems. This is not a fault of C++ language, but of the inadequacy of the `malloc()` routines that in fact might be called internally by C++ allocators (i.e. `operator new`). In such conditions, it is very reasonable to use optimized, custom-memory allocators that are tuned to deal with small memory blocks.

6.4.2. Applicability and conditions

Memory allocation tuning technique is beneficial when applied for applications that frequently allocate small chunks of memory. If an application does not fulfill these two principal conditions – intensive usage of memory and at the same time allocations of small blocks – this kind of tuning is unsuitable. The reasoning behind this is that a single allocation time is relatively small. For example, in the environment where we conducted our experiments a single allocation of 1B of memory done by the standard `malloc()` allocator lasts less than 1 microsecond. Moreover, as we have mentioned standard allocators are not optimized for the allocations of small blocks of memory (tens to hundreds of bytes).

A feature of custom allocators that might be considered a drawback is the earlier usage of larger blocks of memory. For instance, an allocator can obtain large blocks of memory from the general-purpose allocator which it divides into a number of small objects. A custom allocator might also defer object deallocation, returning objects to the system much later than the object's deletion time. In some cases, allocators may never release memory and reuse the allocated blocks until the program is terminated. All these factors may limit the applicability of this technique.

6.4.3. Solution

The solution is to rely on small-objects allocators – specialized allocators that are tuned for dealing with small memory blocks (tens to hundreds of bytes). Small-object allocators use larger blocks of memory so called chunks and organize them in effective way to reduce the total occupied space and decrease total allocation time. There are many custom allocators available, but we chose for this tuning technique a pool-based allocator [Ale00]. This allocator uses a set of variable-length pool (such as vector or list) to support an efficient organization of allocation and deallocation of memory blocks. Each pool consists of an integral number of fixed-size chunks. Each chunk is divided into fixed-size blocks for storing allocated data. When there are no more data blocks available in the last chunk, the allocator creates a new chunk and appends it to the list of chunks. This design is shown in Figure 6.2.

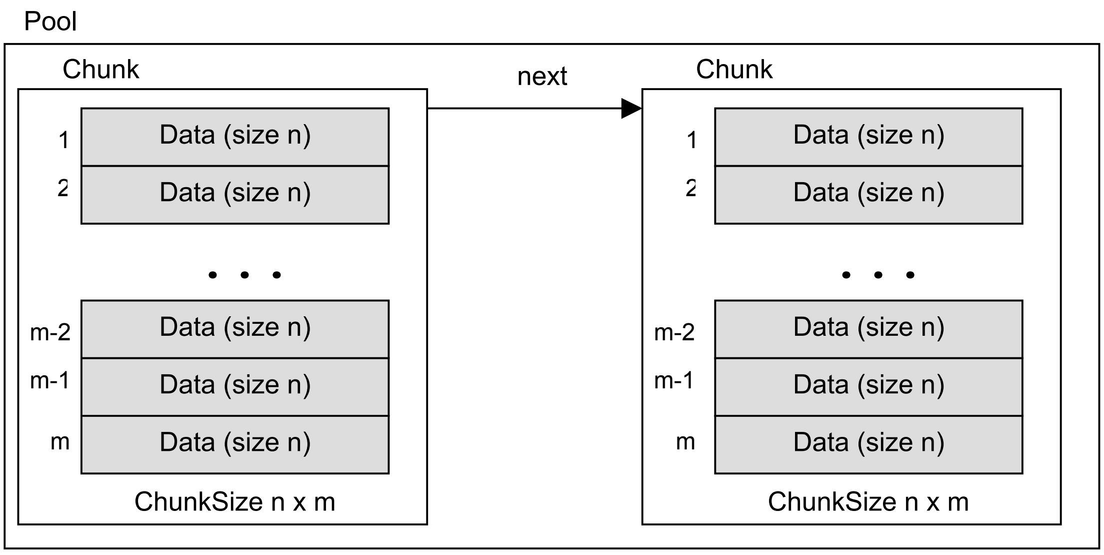


Fig. 6.2. Pool allocator based on the mechanism of fixed-size chunks.

In this approach one pool supports only one size for data allocation; chunk are always divided into m data blocks where each data block has always the same size n. For example, a chunk can has 4KB size and can be divided into 512 blocks for 8B data. If we want to consider allocations of other data sizes, then we have to create a new pool with appropriate characteristics, e.g. a 4KB chunk divided into 256 blocks to fit 16B data.

To optimize the memory allocation performance, we can track the memory allocation requests during application run-time. By monitoring `malloc()` calls we can collect allocation statistics such as histogram of number of allocations and deallocations for groups of request sizes. These histograms show memory allocation usage patterns such as large number of small requests and so on. For example, if we detect thousands of allocations and deallocations of blocks smaller than 4KB, we can conclude that the application is memory intensive and can benefit from optimized pool allocations.

We have determined a set of conditions that activate the tuning procedure for a given range of sizes (`RangeMin, RangeMax`):

- `RangeMax` is smaller than maximum size of the request that is considered small
- number of allocations and deallocations exceeds a threshold `MinAllocs`

If the conditions are met, it is necessary to determined the optimal chunk size for a pool allocator for a given range of sizes. During our investigations we have observed that the particular values of the thresholds should be obtained experimentally.

Finally, we can **replace the standard allocator with the optimized pool allocator for a particular range of request sizes**. Because the individual allocation times are very short (microseconds) we must provide a carefully optimized pool allocator implementation in order to obtain real performance benefits.

6.4.4. Implementation

The implementation of this tuning technique consists of the following components:

- Optimized pool allocator – implemented as a shared library and loaded to the memory of a process by means of `LoadLibrary()` call from the Tuning API. This library provides the following functions:
  - `void * pool_alloc(size_t size)` – this function replaces `malloc()` implementation and delegates some ranges of requests to the internal pool allocator. The other requests are passed to the standard `malloc()` call. The replacement is performed by means of `ReplaceFunction()` call from the Tuning API.
  - `void pool_free(void * mem)` – it replaces standard `free()` call and frees memory previously allocated with the `pool_alloc()` or `malloc()`.
  - `void pool_activate(size_t minRequestSize, size_t maxRequestSize, size_t chunkSize)` – activates the pooling allocator for all memory allocations that request a size in a specified range. The pool will use chunk with the specified size. Invocation of this function is performed by means of `InsertOneTimeFunctionCall()` of the Tuning API.
  - `void pool_deactivate(size_t minRequestSize, size_t maxRequestSize)` – deactivates the pool allocator for a specified range of sizes. The function invocation is performed by `InsertOneTimeFunctionCall()` of the Tuning API.
- Memory allocation tunlet – that is responsible for monitoring the memory allocation requests and determining if and for what sizes pool allocator can behave better than the standard allocator.

When the tunlet is activated, it first requests to instrument the memory allocation routine, i.e. `malloc()`, in order to collect allocation statistics. The tunlet maintains the histogram of number of allocations and deallocations for each range of request sizes, for example sizes smaller than 16 bytes, 32 bytes, 64 bytes, …, 1KB, 4KB, etc. Periodically, the tunlet checks if the number of memory allocations for a given range of sizes exceeds a specified allocation threshold (configurable value) and the corresponding number of deallocations exceeds a deallocation threshold in a given period of time. In that case, the tunlet chooses the best suited size of pool for a given allocation size.

Finally, the tunlet activates the tuning action. This action consists of the following steps:

- If the procedure is invoked for the first time, the tunlet loads the shared library with the pool allocator to a given process. Next, it replaces `malloc()` and `free()` calls with their pool versions using `ReplaceFunction()` method from Tuning API.
- The tunlet invokes the `OneTimeCode()` function in order to call the `pool_activate(rangeMin, rangeMax, chunkSize)` procedure that activates the allocator for a given range.

To replace the allocators at run-time there are several issues to be considered. First, it is necessary to replace `malloc()` and `free()` standard calls with their optimized substitutes: `pool_malloc()` and `pool_free()`. The new functions hold a map of sizes to allocators and in this way decide what allocator (i.e. pool or standard) should be used to handle a request with a particular size. Next issue is related to memory deallocation problem. The problem result from the necessity to match the deallocation request with the allocator that was used to allocate a freed memory block. For example if the memory was previously allocated with the `malloc()` call it must be freed with `free()` call. If it was allocated with a pool allocator, the `pool_free()` function must be used to return the memory to the pool. This is solved by adding extra bytes for each allocated block from the pool. This allows the determination of the appropriate allocator.

6.4.5. Experiments

We wanted to compare the standard C library allocator (`malloc()` and `free()`) to the pool allocator that we have just presented. In order to **investigate the performance of the memory allocations using different allocators**, we have developed a synthetic, C++

program that intensively allocates memory. The application does not perform any processing, but only allocates memory using a specified configuration. We have executed the application using standard allocator and pool allocator with different chunk sizes: 4KB, 8KB, 16KB, 32KB, and 64KB. We used two configurations:

- First, the program has a constant number of allocations and allocates memory for different data sizes. We executed the program for various data sizes ranging from 1 to 65535 bytes and we allocated 1.000.000 times each data size.
- Second, the program applies different numbers of allocations for different data sizes. In this configuration the program was executed for 4B, 8B, 16B and 32B data sizes and for each data size we set different numbers of allocations ranging from 1 to more than 16.000 times.

All executions were conducted in a dedicated scenario, using a single host with no external load.

Measurements and results

Figure 6.3 presents the comparison of average memory allocation time in a function of request size for different allocators: standard and a set of pool allocators with chunk size ranging from 4KB to 64KB. The allocations for each data size were performed 1.000.000 times.

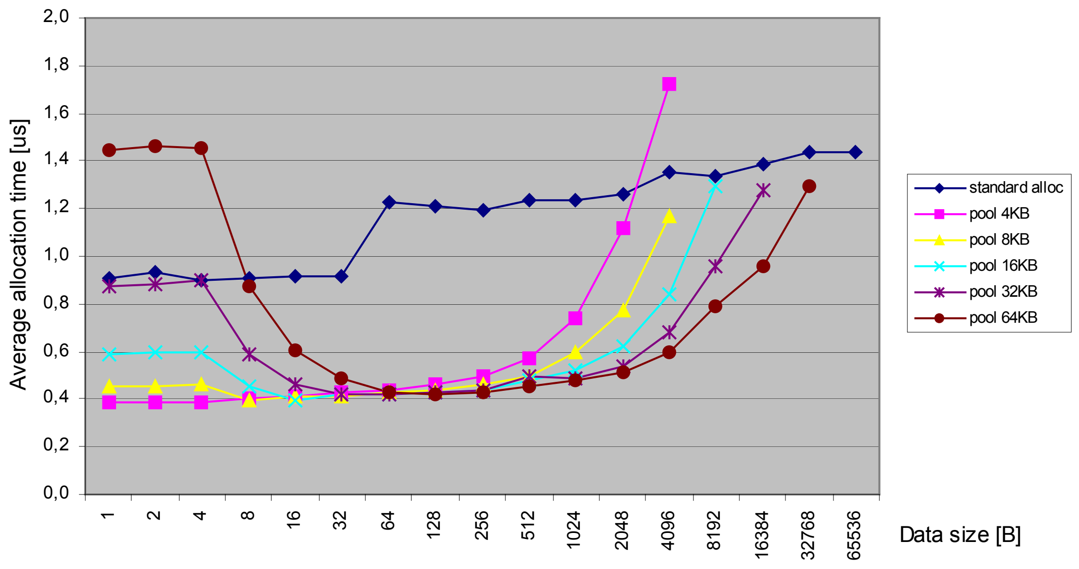


Fig 6.3. Average allocation time vs. data size with constant number of allocations.

We can observe that there is no single optimal allocator for all different request sizes. The optimal strategy would be to use different allocator for different ranges of requests. In general, pool allocators perform better than standard allocator for data sizes smaller than 4KB. The work of the pool allocator is characterized by an expensive first allocation, because the whole chunk must be allocated and divided into blocks. The subsequent allocations up to the end of the chunk are very cheap and with time they may amortize the cost of the expensive allocation. We can notice that different chunk sizes affect the pool allocator performance and there is direct dependency between chunk size and request size. In particular the 64KB chunk performs the best for bigger data sizes but poorly for small requests. The Pool 4KB is the best for small requests, but the worst for big requests.

We can observe that for data sizes equal to 1, 2 and 4 bytes all pool allocators give the constant allocation time. The same happens with the standard allocator but for sizes up to 32 bytes. This can be explained by the memory alignment performed by the allocators. The pool allocators use 4-bytes alignment, and we can guess that the standard allocator groups all request sizes smaller or equal to 32 bytes (i.e. 1 byte request allocates 32 bytes).

Figure 6.4 presents the average time of the free operation for different allocators (with number of deallocations equal to 1.000.000).

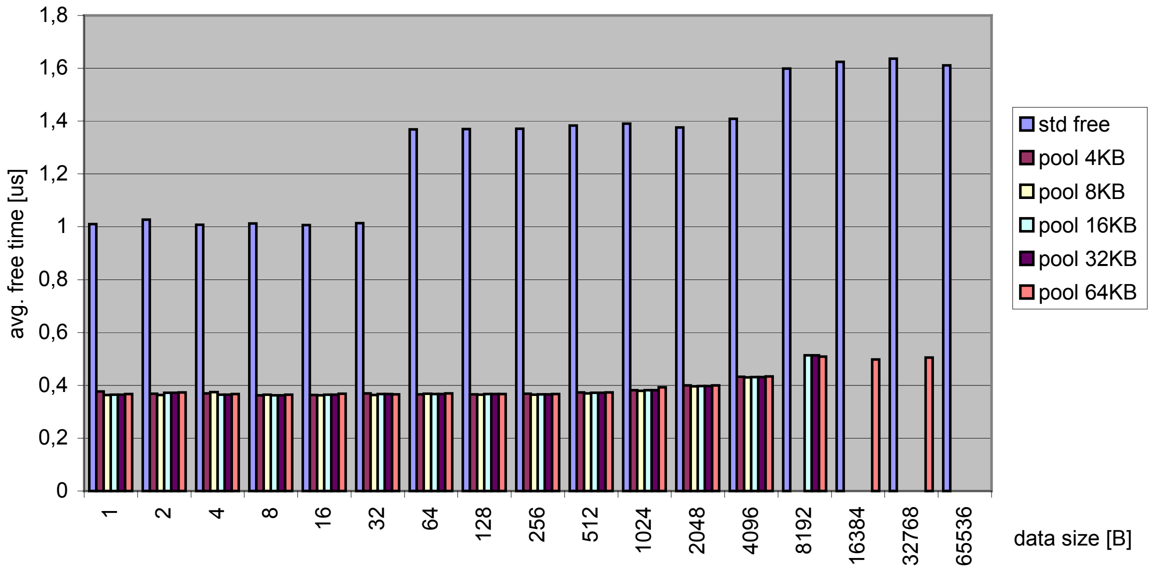


Fig. 6.4. Average deallocation time vs. data size with constant number of deallocations.

We can observe that for all pool allocators the average `pool_free()` time is almost constant and significantly faster (60% - 70%) than standard `free()` function. The constant time results from the nature of the pool – each deallocated memory blocked is simply added to free block list and deallocation does not depend on the size. We can also see that the standard malloc() function groups requests sizes into blocks: sizes ranging from 1 to 32 bytes result in 32 bytes allocations, sizes from 64 to 4KB result in 4KB allocations and so on, because the corresponding `free()` calls have constant times for these ranges.

The following graphs (6.5, 6.6, 6.7, 6.8 correspondently) show average allocation time in function of number of performed allocations for constant requests sizes: 4, 8, 16, 32 bytes.

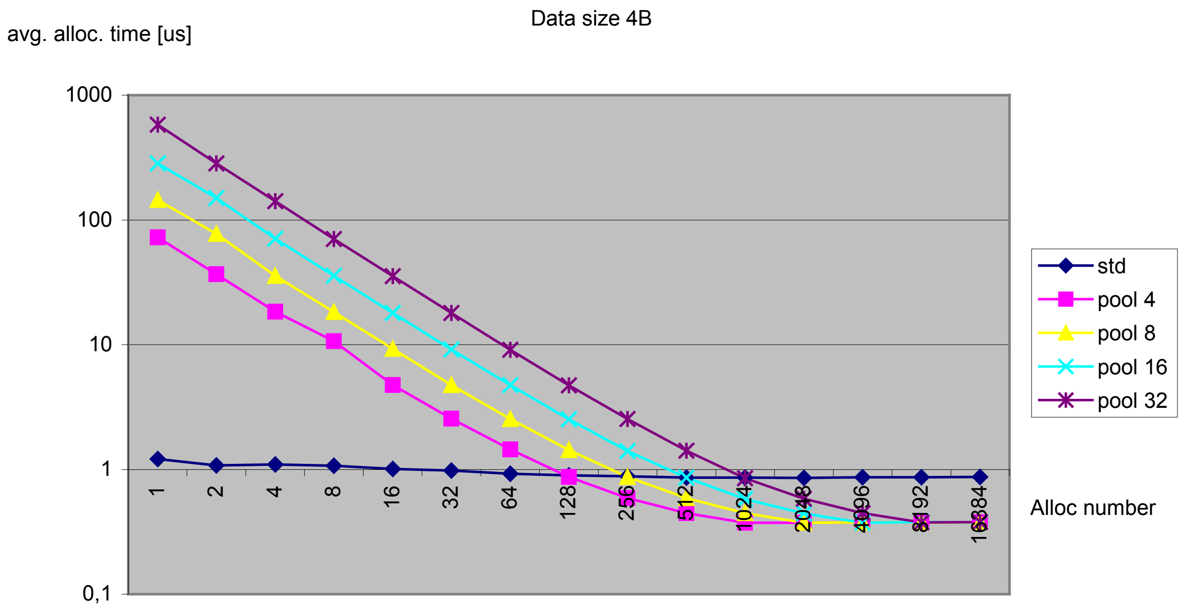


Fig. 6.5. Average allocation time vs. different number of allocations for constant 4B data.

All these figures show the minimal number of consecutive allocations that must be performed by a given pool allocator in order to overcome the performance of the standard allocator. We can observe the bigger data size of the request the smaller number of allocations is sufficient to justify the usage of the pool allocator. In general, we can conclude that number of allocations ranging from hundreds to thousands can be used as a value for the `MinAllocs` threshold.

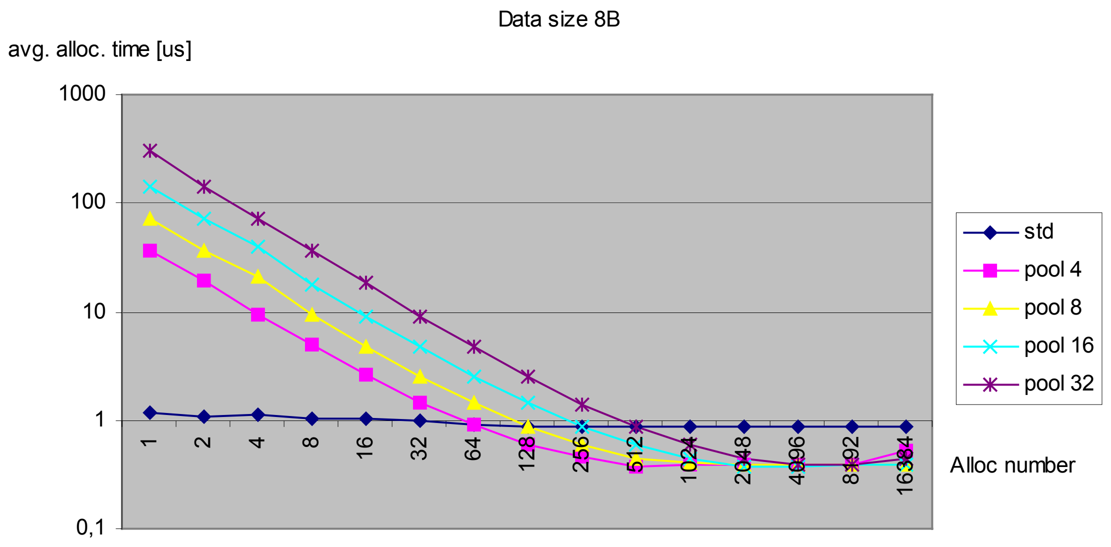


Fig. 6.6. Average allocation time vs. different number of allocations for constant 8B data.

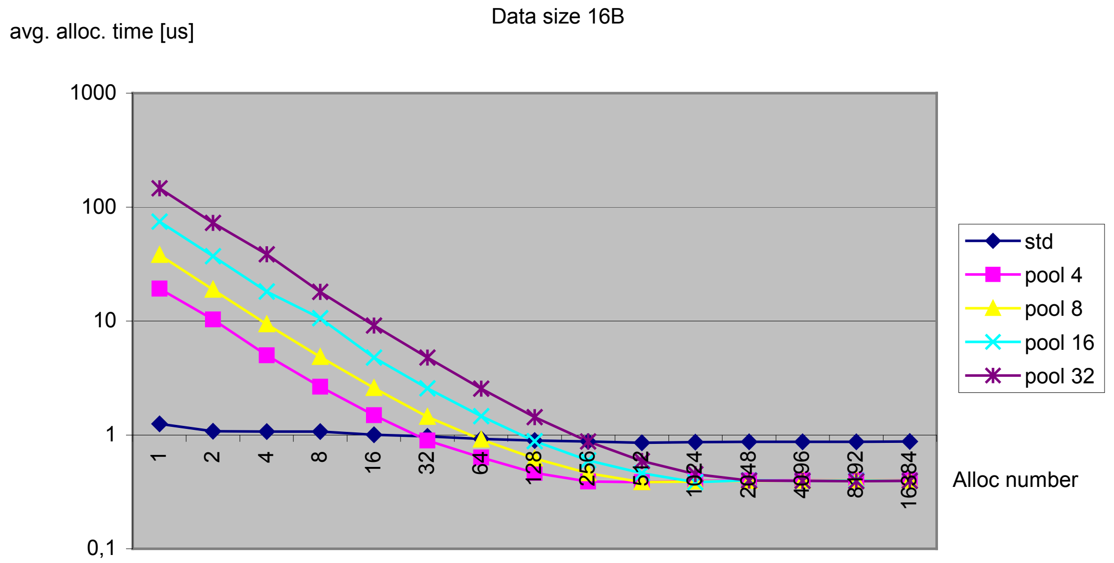


Fig. 6.7. Average allocation time vs. different number of allocations for constant 16B data.

6.4.6. Conclusions

Programs that make intensive use of memory may benefit from optimized pool-based allocators if they perform big number of small object allocations and deallocations. In such conditions, the specialized pool allocators perform much better (up to 70%). However, we have concluded that development of fully automated tuning technique that is able to automatically find out all necessary threshold values and guarantee the performance gain in any conditions is difficult and requires further investigations.

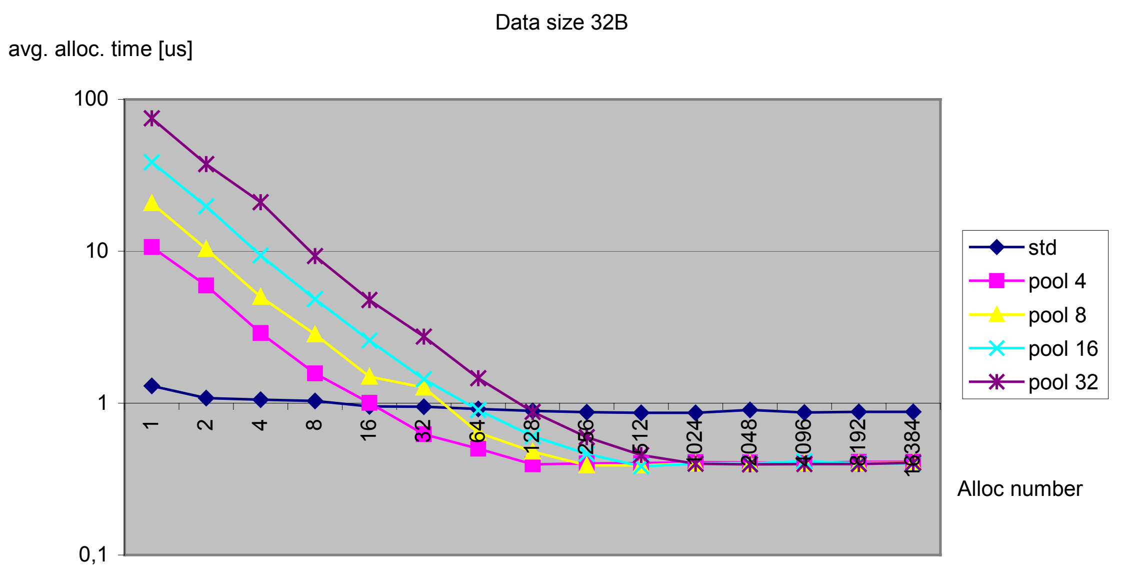


Fig. 6.8. Average allocation time vs. different number of allocations for constant 32B data.

6.5. PVM communication mode

This tuning technique intents to minimize the PVM communication overhead by switching the messaging to the faster, point-to-point mode.

6.5.1. Motivation

The PVM message passing system consists of a daemon process and a set of tasks that use communication primitives. The daemons communicate among themselves using UDP/IP sockets. PVM tasks have the possibility to establish two communication modes with other tasks, the task-to-task mode (called direct) and task-to-daemon-to-daemon-to-task mode (called indirect). By default, PVM uses the indirect communication mode so all the messages exchanged between PVM tasks are routed through the daemons. In the direct mode the tasks bypass the PVM daemon by using direct communication links between them. The direct links are based on TCP/IP sockets and all I/O operations are based directly on system calls (i.e. `read()`, `write()`, `select()`). A given PVM task may have several sockets open at once: one to its local daemon and, optionally, one or more to specific tasks with which it is communicating.

Although the initial TCP set up time is larger, all subsequent communication between the same two tasks is usually faster. This is because the additional routing of each message is avoided. The primary drawback of this method is that each TCP socket consumes one dedicated file descriptor, and in some cases there is a limit on maximum number of opened connections (e.g. some UNIX systems).

PVM provides a clean API to the programmer but the message passing latency is higher than the physical network's latency. In many cases the PVM communication library achieves only 15%- 20% of the network's theoretical capacity [Sub96]. The indirect mode is one of its reasons. Therefore the direct mode, although less scalable, it is a preferred transfer method.

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. Another situation is when the application contains the hard-coded instructions to set the default mode and its source code is not available. Therefore, it is advisable to automatically switch the communication to the direct mode whenever possible during application runtime.

6.5.2. Applicability and conditions

This tuning technique can be applied to all PVM applications that does not explicitly control the PVM communication mode. The direct communication mode is available on majority of the architectures except a few. For example on shared-memory machines, or multiprocessors such as Intel Paragon this mode is not available, because the communication between tasks on these machines always uses the native protocol.

Although communication mode tuning is beneficial even when applied for applications that infrequently exchange small messages, the biggest impact can be seen for communication-intensive applications. This kind of tuning is suitable for problems with rather high communication and rare computation. Therefore this technique should be applied when communication/computation ratio exceeds selected threshold.

An existing limitation is the maximum number of opened connections (i.e. file descriptors) per task. For example some UNIX systems limit this value to 60 per process. So this mode is not available for tasks that use more than 60 file descriptors (i.e. files, sockets, etc.). The advantage is that the PVM automatically switches back to indirect mode, when the direct mode is not available for any reason.

6.5.3. Solution

The application can configure the communication mode explicitly, but by default the indirect mode is used. To set the direct communication mode PVM API offers a function called `pvm_setopt(mode, value)`. This function allows one to modify the PVM library options. First parameter defines which option to set, the second one specifies a new value. The message communication policy option is called `PvmRoute`, and it can have the following predefined values:

- `PvmDontRoute` – do not request or grant connections. This setting on task A sets indirect mode and does not allow other tasks to set up direct links to A
- `PvmAllowDirect` – default value – do not request but allow the direct connection. This setting on task A allows other tasks to set up direct links to A.
- `PvmRouteDirect` – request and allow connections. This setting on task A sets direct links to A. Once a direct link is established between tasks, both tasks will use it for sending messages and it persists until the application finishes.

Function `pvm_setopt()` can be called multiple times during an application execution to selectively set up communication links, but typical use is to call it once after the initial call to `pvm_mytid()`.

During the execution, we can detect the use of indirect mode by calling `pvm_getopt(PvmRoute)` function and check if it is possible to use the 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. One possible complementary variation of this solution is to insert a snippet into the entry of the function `pvm_setopt()` that will modify the value parameter to `PvmRouteDirect` when the mode parameter equals `PvmRoute` for all subsequent calls to that function. Therefore, whenever the function is called, input parameter value is ignored and always set to indicate direct communication mode. In that way the application will not be able to change the mode back to indirect until the snippet is removed.

6.5.4. Implementation

To monitor what communication mode is used and if it is changed explicitly by the application, a snippet is inserted into the `pvm_setopt(mode, value)` function to each individual task. To receive the event record, the tunlet must register appropriate callback `Event::SetEventHandler`. Each time the function `pvm_setopt()` is called, the tunlet receives a corresponding event record that contains the parameters of the function (i.e. mode and value). For example the tunlet receives a notification when a task activates the direct mode by calling `pvm_setopt(PvmRoute, PvmRouteDirect)`.

The tunlet that implements this technique must be able to verify the configuration of the virtual machine. This is possible by handling notifications related to addition and removal of tasks and hosts – callbacks `Application::SetTaskHandler`, `Application::SetHostHandler` (see Chapter 5 Dynamic Tuning API for more details). The analysis model for this technique is based on simple rules. If there are no shared-memory machines and number of tasks does not exceed the system-dependent limit (i.e. 60 file descriptors) the direct mode is considered available. In that case, for each task that uses the indirect mode, the tuning action can be applied. If a task already uses direct mode, there is no need to apply any tuning action.

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 tuning action must be synchronized with the application execution. For example, if a task is executing the code inside the `pvm_send()` function, the inserted invocation of the `pvm_setopt()` action may provoke the communication failure, because the modification of the communication mode changes the underlying sockets. Therefore, first the breakpoint is inserted at the entry of function `pvm_send()` and when it is activated, the actual invocation is performed.

6.5.5. Experiment 1

In order to **compare the communication performance of both PVM communication modes**, we have developed a simple, synthetic, PVM master-worker program that exchanges messages in a ping-pong manner. The master task sends a determined amount of work and waits for results. The worker receives a work, and immediately sends the same amount of work back to the master. There is no computational operations, the application is strictly based on communication. We executed the program for various message sizes ranging from 1 to 1.000.000 bytes. At startup, the master task configures the selected communication mode and later whole communication is performed in that mode.

We have performed the experiment for both direct and indirect communication modes. All executions were conducted in homogeneous and dedicated scenario (no external load). In this scenario, we have used two homogeneous machines (aows6, aows7) connected by 100Mb/sec network. We have called this scenario dedicated because we tried to minimize the possibility of external influences executing the application when all workstations were idle. However, as described before, the cluster was not physically isolated from the network and in fact could be accidentally used by other users.

Measurements and results

Figure 6.9. presents the comparison of communication performance between PVM direct and indirect communication modes in the ping pong application presented above (note that both scales are logarithmic). The detailed measurements are listed in Table 6.3.

As expected, we can observe that independently on the message size the change from indirect to direct mode results in significantly faster communication in all cases (up to 50%). In particular, the big difference between both modes is noticed for small messages when the startup time highly influences the transmission time. The startup time is constant and hence it is very notable for small messages when the transmission time is small, in opposite it may be almost lost for big messages when more time is required for providing a message to the destination. In the case of indirect mode the additional message routing task-to-daemon-to-daemon-to-task is performed and at each point the startup time is added (3 times, namely from task to pvmd, from pvmd to pvmd on the remote destination machine, from that pvmd to destination task). In the direct mode, the startup time influences only once (from task to task). For small messages, the benefits can reach even 50% The bigger a message is, the less the benefits are, but for all presented message sizes we see the direct mode profitable, e.g. for 1MB message – 18% of profits.

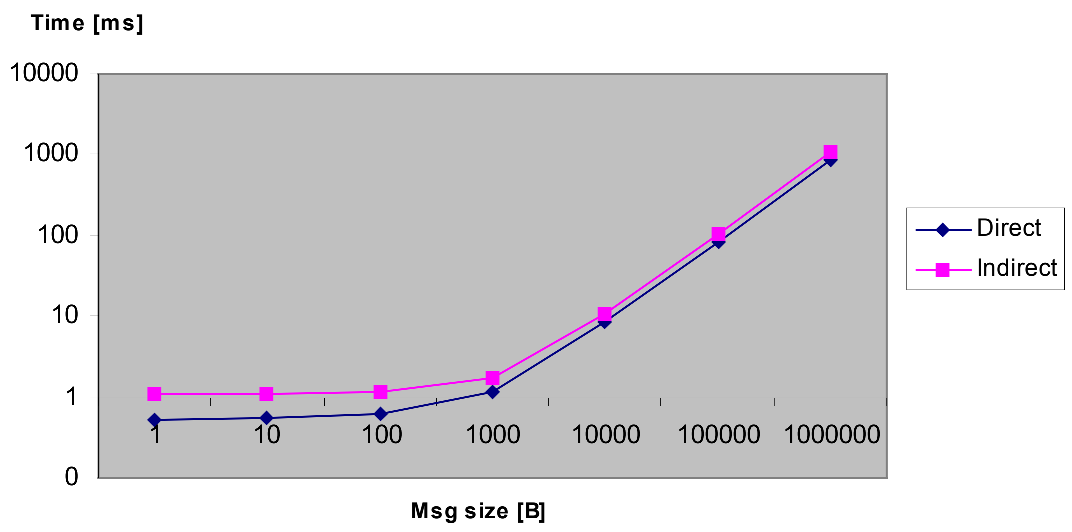


Fig. 6.9. Benefits gathered from changing communication mode in a ping-pong application (logarithmic scale).

Moreover, the cost of additional indirect routing is presented since the PVM divides a bigger message into a set of fixed-size fragments before sending it to the destination. By default the PVM library uses a fragment size of 4KB. Additionally, internally PVM is based on the socket communication and uses constant packet size of 32KBytes. Therefore, if bigger message is sent, it must be divided in more fragment size and in more socket packets. We describe the message fragment size in Section 6.7.

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%

Table 6.3. The detailed measurements of PVM communication performance for direct and indirect communication modes.

6.5.6. Experiment 2

The goal of the next experiment was to **compare the performance of a real application applying tuning of the PVM communication mode**. To conduct our experiments, we selected a communication-intensive parallel program. We used **Integer Sort (IS) kernel benchmark from NAS** [L40] **Parallel Benchmark suite** [Bai94, Bai95]. The IS kernel ranks a large array of small integers as fast as possible using a bucket-sort method.

Bucket sort [L41] is the fast sorting methods because it does not perform any key comparison. However, there are significant limitations in its usage and it can be applied sufficiently only in rare situations. To do a bucket sort, a temporary array must be used in which the elements to be sorted are distributed basing on their key fields. If the maximum key value in the list is n, then the temporary array should be at least of size n+1. For example, if we must sort 2, 9, 6, 5, 1, 7, the temporary array should be at least of size 10. To distribute the numbers in the temporary array, this array is first initialize with a flag value – a value that cannot be a key field of the sorted numbers, e.g. –1 in the above example. Next, each element with a key field n is copied in position n of the temporary array. Finally, all non-flag values from the temporary array (i.e. numbers with value different than –1) are copied back into the original structure in the order they appear in the temporary array. The distribution of n numbers requires n steps and thus, the performance of bucket sort is of order n (linear).

Bucket sort works only under very restrictive conditions. These are:

- The key field must be unique positive integer and not string, float or even negative integers.
- The range of values for the key-field must be relatively small, otherwise, the temporary array will be too large. E.g. if the key field is ID number (6 digits), then the temporal array must contain a place for 999999 elements (which is impossible to store in the memory).

The IS NAS benchmark is based on the master-worker paradigm. The main program (master) generates a vector of integer data (keys) to be sorted using the pseudorandom number generator. The keys are in the range [0, max_keys) and distributed as a Gaussian. Gaussian distribution is also know as normal distribution [L42] and it is a family of distributions that have the same general shape. They are symmetric with scores more concentrated in the middle than in the tails. This distribution can be also described by a symmetric bell-shaped curve. The keys to be sorted are performed according to the following scheme. The master task divides all existing keys in *number of keys / number of* *nodes* parts. Each part must be distributed to one worker. First, the master sends to each worker a message with the information that specifies the range and number of the keys. Then, it sends a data. A worker receives data from the root node and samples it to arrive at a good load balance. It communicates with other workers in order to know their ranges. Then it keeps all keys which fall in its range and sends other keys to the appropriate nodes. Finally, it sorts all the keys in its range.

The IS NAS kernel tests both integer computation speed and communication performance. Communication costs are high (up to about 50% of communication) in this application. This is because the benchmark is dominated by all-to-all data exchange messages, since each processor sends to all others that data which falls within the range of the recipient.

All executions performed with IS NAS benchmark were conducted in the same environment as in the previous experiment described in Section 6.6.5 (homogeneous and dedicated). The only change we done was the number of machines incorporated into the PVM virtual machine. We were executing the IS NAS benchmark on 4 machines (aows1, aows6, aows7, aows8).

Measurements and results

Table 6.4 presents the results of the IS kernel benchmark experiments in two different tuning scenarios. In the first scenario, the application was executed under standard PVM 3.4 without any tuning. In the second scenario, the PVM communication mode was monitored, analyzed and optimized by MATE. The tunlet that was responsible for the PVM communication mode decided to use the direct mode as all required conditions were accomplished. By default the application used indirect mode and our experiments were conducted in a small NOW environment. We can observe a 17,5% benefit in execution time caused by this tuning action. Such an improvement can be explained by a high computation-communication ratio (1:1). As we have mentioned in the Applicability part, this tuning technique is adequate for communication intensive applications. The IS NAS benchmark spends up to about 50% of the execution time on the communication and it is very profitable to avoid the additional routing caused by the indirect mode. The measured intrusion did not exceed 3,5% of the total execution time. We see then that in this case tuning is effective and benefits are higher than the overhead introduced into the application execution.

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%)

Table 6.4. The measurements of PVM communication performance when applying dynamic tuning of communication mode in the IS NAS benchmark.

6.5.7. Conclusions

This tuning technique results beneficial for applications with high communication-computation ratio that do not explicitly control the PVM communication mode. The tuning cost is very low, because it is simple to obtain necessary information and the tuning action has a cost of a breakpoint and a single function call. However, this technique has rather static nature, because the change is typically performed once during execution of the task. So if the application source code is available it might be reasonable alternative for the programmer to modify the source code and recompile the application.

6.6. PVM encoding mode

This tuning technique intents to minimize the PVM encoding overhead by skipping data encoding/decoding phase.

6.6.1. Motivation

An application running in the PVM environment, can be executed on different machines that form PVM virtual machine. These machines do not have to be homogeneous, because the PVM system transparently handles all necessary operations for processing in a heterogeneous, network environment, e.g. inclusion of heterogeneous machines to the virtual machine, message routing between all machines, data conversion for incompatible architectures. To be able to route messages in a heterogeneous environment the PVM library must apply specific mechanisms. One of them is data format conversion. When PVM transfers the data, it must convert the data format transparently between machines that have different architectures. It is achieved by using External Data Representation (XDR) standard.

XDR [L43] is an encoding of simple and aggregate data types that enables exchanging information between different systems and programming languages. In order to transmit data between nodes, first, a data format is translated from the internal data format of the sending node to the XDR encoding (machine-independent format). Then, the XDR encoded data is sent over the network to a destination. Finally, the receiving node translates the data from the XDR encoding to its native representation.

The disadvantage of XDR is that the data is necessarily bigger because of the additional information that permits to read the data correctly. In XDR the representation of data requires a multiple of four bytes (or 32 bits). The bytes are numbered 0 through n-1. The bytes are read or written to some byte stream such that byte m always precedes byte m+1. If the n bytes needed to contain the data are not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, to make the total byte count a multiple of 4. The XDR format must support efficiently different machine architectures and not cause the memory alignment problems. Therefore, the size of principal XDR unit has a value 4. For example, in the case of short integer type, in XDR it always consists of 4 bytes, not of 2. In the case of a string, appropriate number of 0s will be added at the end of the string to make its length divided by 4. Moreover, the string length (4 bytes) is added at the beginning in the way that byte 0 of the string always follows the length.

Another drawback is caused by additional processing that is needed on both sides, the sender and the receiver. The sender must include the additional information (encoding) and the receiver must interpret this information in order to translate the encoded data (decoding). In the case of integer type, the most and least significant bytes are 0 and 3, respectively. The XDR encodes integers in big-endian byte-order. If sending and/or receiving machine has an architecture that supports little-endian order each integer then must be translated.

By default PVM encodes data using XDR standard, because it cannot know if the user is going to add a heterogeneous machine before a message is sent. If there is no heterogeneous machine in the PVM virtual machine or messages are exchanged between tasks on the same 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 reduction of sending data size, avoiding data encoding/decoding costs and in result reducing execution time.

The application may contain the hard-coded instructions to set the default encoding mode since a developer does not know in what environment an application will be executed. Moreover, its source code may not be available. Therefore, it can be beneficial to automatically switch off the default XDR encoding whenever possible during application runtime.

6.6.2. Applicability and conditions

This tuning technique can be applied to all PVM applications. The applicability of this tuning technique mainly depends on the architecture of the machines included into a virtual machine If the virtual machine consists of machines with compatible architectures, then the XDR encoding can be avoided. Moreover, if the whole application is running only on one machine or there are at least two processes on one machine that explicitly communicate among them, then the data raw mode can be used. In the opposite cases, the XDR encoding must be applied.

It must be pointed out that the encoding mode tuning, similarly to the communication mode tuning, is beneficial when applied for applications that frequently exchange messages. It is suitable for communication-intensive applications. If an application does not exceed selected thresholds of communication/computation ratio, there is no sense to tune the encoding mode. In such a case, the influence of the encoding phase is not significant as there are few and/or infrequent messages to be encoded.

Another issue to be mentioned in the case of encoding tuning is the type of exchanged data. Each data type must be encoded to the XDR format to be machine independent. However, the time consumed to encode one type may differ significantly from the encoding phase of other types. For example, integer or float data encoding causes big amount of time lost on the transformations. Each integer must be encoded according to the big-endian byte-order. Each float is represented according to a floating-point numbers standard [IEE85]. String type encoding is not time consuming, since there is no transformation of the string itself. XDR only adds the maximum of 7 bytes to it (4 bytes representing the length, 0-3 bytes to make the total byte count a multiple of 4). The encoded string can be then sent via the network in the comparable time as the string in data raw format. Summarizing, the tuning technique is more applicable for the data types that required many transformation (integer, float) and/or significant number of bytes is added to the data (short integer).

6.6.3. Solution

The application can configure the encoding mode explicitly. Before any message can be sent, function `pvm_initsend(encoding)` must be invoked. This function clears the default send buffer and sets the message encoding mode. It packs all data into a message in one of several encoding formats. PVM predefines five sets of encoders and decoders. The two most commonly used ones pack data in raw (`PvmDataRaw` – host native) and default (`PvmDataDefault` – XDR) formats. Specifying the `PvmDataRaw` value indicates that no data conversion should take place. `Inplace` encoders pack only descriptors of the data (pointers to static data), so the message is sent without copying the data to a default send buffer. There are no inplace decoders. `Foo` encoders use a machine-independent format that is simpler than XDR; these encoders are used when communicating with the pvmd. `Alien` decoders are installed when a received message can't be unpacked because its encoding doesn't match the data format of the host. A message in an alien data format can be held or forwarded, but any attempt to read data from it results in an error.

In this work we focus on tuning the default encoding mode called `PvmDataDefault` and changing it to the `PvmDataRaw`. During the execution, we can detect the use of encoding mode by inserting a snippet into the entry of the function `pvm_initsend()` and check if it is possible to use the data raw encoding mode. This mode is available when all machines from the PVM virtual machine have the same, homogeneous architecture. The tuning action includes the insertion of a snippet into the entry of the function `pvm_initsend()` that will modify the value of its unique parameter `encoding` to `PvmDataRaw`. Therefore, whenever the function is called, input parameter value is ignored and always set to indicate data raw encoding mode. In that way the application will not be able to set the encoding mode until the snippet is removed. Although the function `pvm_initsend()` is invoked setting data raw mode, PVM verifies the machines' homogeneity when the `pvm_send()` call is made. If two machines that exchange the message do not have the same architecture, PVM aborts the transfer of the message.

6.6.4. Implementation

To monitor what encoding mode is used, a snippet is inserted to each individual task into the function `pvm_initsend(encoding)`. Each time this function is called, the tunlet receives a corresponding event record that contains the parameter of the function (in this case the encoding mode). To be able to receive event records, the tunlet must register appropriate callback `Event::SetEventHandler`.

The application can configure the data encoding mode explicitly, but the most common one is `PvmDataDefault` – XDR encoding mode. The MATE environment controls the addition of hosts to the virtual machine (via hoster service) so it is able to detect if an application is executed in the homogeneous cluster. The tunlet that implements this technique must be able to check the configuration of the virtual machine. This is possible by handling notifications related to addition and removal of tasks and hosts – callbacks `Application::SetTaskHandler` and `Application::SetHostHandler` (see Chapter 5). The analysis model for this technique is based on simple rules. During the execution the tunlet can decide to use data raw mode if the condition of hosts' homogeneity is fulfilled. The tuning action should be applied only on the tasks that use the function `pvm_initsend()` and send messages.

The tuning action includes insertion of instrumentation (entry of `pvm_initsend()`) that changes the encoding mode from XDR to data raw (parameter mode is always set to `PvmDataRaw`). In this case no synchronization is required, because the tuning action will be invoked only when the application reaches the `pvm_initsend()` entry. Moreover, when a new machine with different architecture is about to be added, the Analyzer module can request to restore the XDR mode.

6.6.5. Experiment 1

To **compare the encoding performance of both PVM encoding modes**, we have adapted a synthetic, PVM master-worker program that was used for purposes of the comparing the communication performance (see Section 6.5 PVM communication mode). This program exchanges messages in the same manner as described above and we executed it for the same various message sizes ranging from 1 to 1.000.000 bytes. At startup, the master task configures the encoding mode and later whole communication is performed in that mode. We have performed the experiment for both XDR and DataRaw encoding modes. All executions were conducted in the same environment as in the case of communication mode (homogeneous and dedicated, aows6, aows7).

Measurements and results

Figure 6.10 contains the comparison of PVM encoding performance. It shows XDR vs. data raw encoding modes applied in the ping pong application presented above (both axis X and Y are logarithmic). Additionally, we present the detailed measurements obtained from the experiments in Table 6.5.

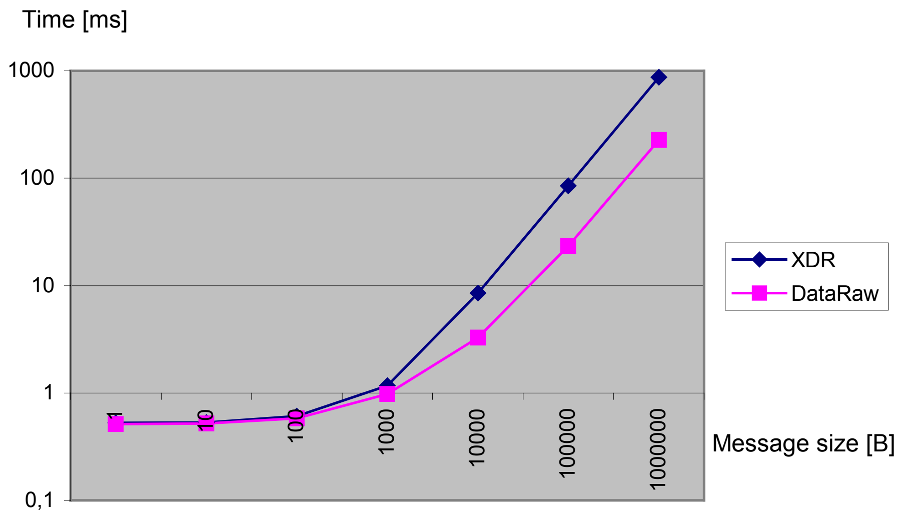


Fig. 6.10. Benefits gathered from changing encoding mode in a ping-pong application (logarithmic scale).

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%

Table 6.5. The detailed measurements of PVM encoding performance for XDR and DataRaw encoding modes.

We can observe the significant benefits obtained when executing our benchmark application in different modes. Changing XDR to data raw encoding mode, we may achieve up to ~74%. We can see that XDR encoding overhead grows together with the message size. For message sizes less than 1KB, the difference is low (about 2% - 4%) and hence not so significant. If bigger amount of data is sent, more time is required for encoding and decoding it. We conclude then that the data raw mode is significantly faster and preferable in the typical homogeneous clusters.

Moreover, we must pointed out the data types issue. Figure 6.10 and Table 6.5 show the results of the encoding performance for messages that contain only integer data. The tested application allocates the array of integers and this array is then exchanged between tasks. To compare how different data types behave when encoded, we executed the same benchmark application modifying data type of exchanged messages.

Figure 6.11 shows the differences in the application execution times when applying encoding on float and string (set of chars) data types. The detailed results are listed in Table 6.6 and 6.7 for messages with float data and string data, respectively. We can notice that XDR encoding has a big influence into the float data type, similarly to the integer data type. The benefits from applying data raw mode can reach up to 64% for big messages. In the case of integers, we obtained up to 74% of profits. It means that the integer transformation to XDR format consumes even more time than for float data.

When exchanging messages consisted of the string data, although we change the encoding mode from XDR to data raw, there are no real differences in the transmission time. The application execution times for both modes, XDR and data raw are comparable. As we could suppose, such a phenomena takes place as the string type data does not require significant transformations (we mentioned above that a string itself is not encoded, only maximum of 7 bytes is added). The times of exchanging strings are also similar to the messages with integer and float data exchanged in data raw mode (Table 6.5 and Table 6.6 correspondingly). The small differences between all these execution times appear only because of our experimental environment in which nor the cluster nor interconnection network was completely dedicated and isolated to purpose of the experiments.

MsgSize [B]	XDR Time [ms]	Data Raw Time [ms]	Average Benefit %
1	0,56 (±0,02)	0,55 (±0,03)	1,08%
10	0,54 (±0,03)	0,55 (±0,03)	-2,30%
100	0,64 (±0,03)	0,62 (±0,03)	3,10%
1000	1,18 (±0,03)	1,00 (±0,03)	15,45%
10000	6,75 (±0,14)	3,34 (±0,11)	50,50%
100000	67,68 (±2,7)	26,6 (±2,53)	60,69%
1000000	690,62 (±40,16)	246,29 (±25,98)	64,34%

Table 6.6. The detailed measurements of PVM encoding performance for XDR and DataRaw encoding modes when exchanging floats.

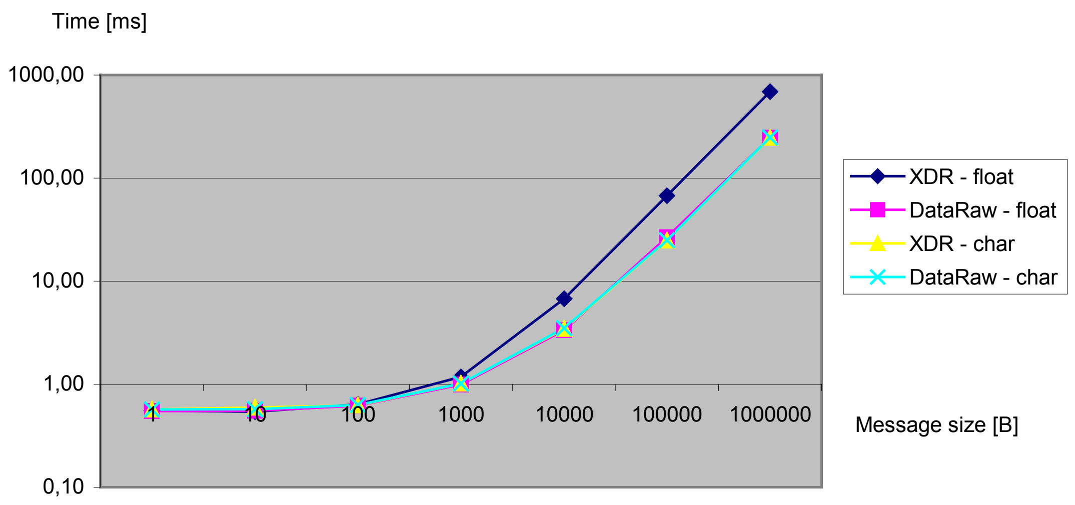


Fig. 6.11. Comparison of the PVM encoding performance in a ping-pong application for different data types of exchanged messages, float and chars (logarithmic scale).

MsgSize [B]	XDR Time [ms]	Data Raw Time [ms]	Average Benefit %
1	0,57 (±0,02)	0,57 (±0,02)	-0,22%
10	0,59 (±0,03)	0,57 (±0,03)	3,21%
100	0,63 (±0,03)	0,63 (±0,03)	-0,68%
1000	1,02 (±0,03)	1,02 (±0,03)	0,11%
10000	3,44 (±0,1)	3,49 (±0,09)	-1,65%
100000	24,72 (±2,22)	25,01 (±1,96)	-1,16%
1000000	246,18 (±24,45)	248,18 (±25,92)	-0,79%

Table 6.7. The detailed measurements of PVM encoding performance for XDR and DataRaw encoding modes when exchanging strings.

6.6.6. Experiment 2

In order to **compare the encoding performance applying tuning of the PVM encoding mode in a real application**, we conducted experiments on the same application as in the case of PVM communication tuning. We used **Integer Sort (IS) kernel benchmark** from NAS Parallel Benchmark suite which was described above. All its executions were performed in the same as above environment (homogeneous and dedicated using 4 machines: aows1, aows6, aows7, aows8).

Measurements and results

Table 6.8 shows the results of the IS kernel benchmark experiments in four different tuning scenarios. The first scenario presents the original application execution under standard PVM 3.4 without any tuning. In the second scenario, tuning the data encoding mode was tried. In this case the tunlet responsible for this optimization did not perform any tuning actions because the required conditions were not accomplished to activate them. It was caused by the settings performed in the application. Originally the application already used the data-raw encoding mode, so no improvement was possible. PVM encoding mode tuning caused in this case the longer application execution time (for about 3,9%). It was caused by the intrusion inserted by MATE into the execution, namely application startup (task load time, task image parsing time), instrumentation (adding, execution and reporting) and analysis. The intrusion reaches up to 2,8% of the total application execution time.

No.	Tuning scenario	Execution time [sec]	Tuning benefits [sec]	Intrusion [sec]
1.	PVM + DataRaw (original execution, no tuning)	732	-	-
2.	PVM + DataRaw + tuning (wrong decision)	757	-29 (-3,9%)	21 (~2,6%)
3.	PVM + XDR (no tuning)	1432	-	-
4.	PVM + XDR + tuning	759	637 (47%)	21 (~2,8%)

Table 6.8. The measurements of PVM communication performance when applying dynamic tuning of encoding mode in the NAS IS benchmark.

However, to see what would happen if the application was executed in the XDR mode we experimentally run it in this mode. We can observe in the scenario 3 that application execution time dramatically increased comparing to original time (in XDR 1432sec, in DataRaw 732sec) – about 49% longer. Then, in scenario 4, we run the application in the XDR mode but it was tuned by MATE. The improvement caused by the tuning action attained 47%. Such benefits were obtained because of two factors. First, the type of the application – it is the communication intensive application. We mentioned that this tuning technique as focuses on the communication time optimizations is suite for programs that communicate a lot. The second factor has the originality in the exchanged data. IS NAS benchmark serves for sorting integer numbers, and hence the high volume of messages exchanged between tasks contains the integer data.

6.6.7. Conclusions

This tuning technique can improve the communication performance between two processes that are running on machines with the same architecture by eliminating the redundant encoding. The technique is suitable for communication-intensive applications and brings benefits in homogenous VM configurations where encoding is unnecessary overhead. The benefits are dependent on the amount of exchanged messages and data types. The technique adapts the application to dynamic changes in the environment, because it selects the optimal encoding mode whenever VM configuration changes during execution of PVM-based applications.

6.7. PVM message fragment size

This tuning technique intents to choose the optimal size of message fragments to minimize the PVM communication time.

6.7.1. Motivation

In PVM messages exchanged by the tasks are composed without a maximum length. Before a message is physically sent, it must be prepared putting all data into the active send buffer. This is performed by calling any of the family functions `pvm_pack()`. The pack functions allocate memory in steps and store messages. Internally a message is put into a fixed-size data buffer. This buffer is called fragment. PVM uses a default fragment size of 4KB. If a message is bigger than 4KB and hence does not fit one fragment, it is divided into appropriate number of fragments. Then the pack functions allocate additional memory to put there the rest of the fragments. Each fragment is separately sent. When sending large messages, a number of fragments must be allocated and then separately sent. PVM uses specific data structure to manage the fragments and chain the them together into a list. Although underlying services might have lower limits, PVM implementation limits the maximum fragment size to 1024KBytes.

Internally, PVM uses sockets for communication purposes. Socket communication is packet-based (data to be sent is divided into packets) and has certain limitations concerning the packet size (see Section 6.3 for TCP/IP buffers tuning). In different architectures the default and maximum settings can differ. In our case (Unix Sun Solaris) by default the maximum packet size is set to 32KBytes. PVM sets the packet size to this value and this is unchangeable during the application execution.

Such a message division provokes the following situation: a message before it can be sent is divided into PVM fragments, then each fragment is put into socket packets. This causes high fragmentation what may reduce performance. Therefore, although having constant socket packet size set to 32KBytes, 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.

As we have mentioned, by default PVM uses 4KB message fragment size. If an application does not explicitly change this value or sets it to a specific one, such a situation might cause non optimal application behavior. Moreover, the fragment size depends deeply on the message size. When a message size varies during the application execution, the message fragment size should also be optimized for the existing conditions. We experimentally deduced that the optimal fragment size depends on application behavior – size of data sent and received. The optimal value can vary during execution (due to program phases) and thus it is not enough to calculate it once; it should rather be adapted to the application behavior. Additionally, an application source code may not be available. For all these reasons, the automatic and dynamic tuning of the message fragment size can be beneficial.

6.7.2. Applicability and conditions

The drawback to this strategy is increased memory usage. If the message fragment size must be increased, more memory is required since PVM allocates additional and bigger buffers for the exchanging messages. The PVM implementation has a problem related to memory allocation. When memory is allocated for buffers for a specific fragment size there is no possibility to free memory already allocated for other size. For example, by default the 32KB buffers are allocated. When the fragment size is changed to 512KB, PVM allocates appropriate buffers, but none of the 32KB buffers is freed (they remain in the static buffer pool until the end of the program). Therefore, when the fragment size is often changed the whole available memory may be consumed and no space will be left for new allocations. One point must be mentioned here. If the fragment size is changed to the size that was used before (e.g. back to the 32KB considering the example above), PVM utilizes the memory that was already allocated for this size. In this case the message packing may be faster since no new allocations are performed (except the case when a message does not fit into allocated fragments). However, in general such an approach provides to quick resource exhaustion. Therefore it is necessary to limit the number of changes with a counter. If the message fragment size was changed n times to different sizes, then no fragment size tuning should be invoked any more. This decision can be followed by the appropriate instrumentation removal, un-registration of callbacks and the end of the tunlet work.

It must be also pointed out that fragment size tuning is not very effective in indirect communication mode, because it is only performed on the application tasks and does not affect the behavior of PVM daemons.

As it was in the case of PVM communication and encoding mode tuning, it is reasonable to apply this tuning technique for applications that intensively exchange messages. Only if an application exceeds selected thresholds of communication/computation ratio, the performance improvement can be noticeable. Otherwise, intrusion introduced by the dynamic tuning operations might cause the performance deterioration.

6.7.3. Solution

It is possible that the application configures the PVM message fragment size explicitly, but by default the determined size is used – 4KB. To set the message fragment size, the PVM library provides a function called `pvm_setopt(mode, value)`. This function has been also described in Section 6.5.5 while presenting the tuning technique for PVM communication mode. However, this function is more general and serves to set more PVM library options. The mode value can be set to `PvmFragSize`, what means that the function changes the size of PVM message fragment. The second parameter will have a value of the size in bytes.

When the application is running, we can check the current PVM message size, and hence see if it was changed explicitly by the application by calling `pvm_getopt(PvmFragSize)` function. To choose the optimal value of the message fragment size, we must know the sizes of the messages that are exchanged between the tasks. Moreover, we should be able to obtain conditions of the application and environment in which it is running as we must reflect all conditions related to the message fragment tuning applicability. Basing on this knowledge, we have to conclude the activating conditions for fragment size calculations and tuning action and moreover, the dependency of the fragment size from the message size. There is no specific already-provided mathematical model for this purpose and hence we must experimentally determine such a model.

To start the new fragment size calculation, first we must check the conditions. The application must be executed in direct mode (see Section 6.5 for more details). A special pattern must occur – high frequency of messages with size > 4KB. Memory is available and it is possible to change the fragment size (a number of changes did not reach the maximum). If these conditions are true, then we can calculate the optimal fragment size. Currently, to calculate it, we use the experimentally deduced formula:

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

This formula gives a balance between the very small and very large messages. To calculate the optimal value, we must gather the number and sizes of exchanged messages. The information about the number of sent messages can be obtained by instrumenting communication functions (i.e. `pvm_send ()`). Together with the number of messages, we can capture their sizes. It is performed in two steps. First, the function `pvm_getsbuf()` must be called to return the identifier of the message buffer where the sent message was put. The second step is to call the function `pvm_bufinfo()` that returns information about a specified message buffer. This information contains the length in bytes of the entire message, the message label (tag) and the source of the message.

The optimal fragment size can be calculated and changed according to the given performance model. If the optimal fragment size is different from the current one, the tuning action must be performed, but only under certain conditions. The action is not applied each time the new optimal value is calculated. Instead, the tuning action should be triggered when the difference between current and optimal values exceeds a fixed threshold, and the estimated communication cost becomes significant.

The tuning action includes one-time invocation of the function `pvm_setopt(PvmFragSize, size)` that changes the PVM fragment size to a given size. In this case, similarly to the PVM communication mode tuning, we can apply also complementary solution. It may insert a snippet into the entry of the function `pvm_setopt()` that will automatically set the value to `size` when the parameter equals `PvmFragSize`. If the fragment size has been already set to a specific value, the next time it must be changed, the tuning action must be removed and inserted once again setting the fragment size to the new value.

6.7.4. Implementation

The tunlet that implements this technique must be able to check the configuration of the virtual machine. This is possible by handling notifications related to addition and removal of tasks and hosts – callbacks `Application::SetTaskHandler`, `Application::SetHostHandler` (see Chapter 5 Dynamic Tuning API for more details).

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()`. To receive event records, the tunlet registers the callback `Event::SetEventHandler`. When the function `pvm_setopt()` is called, the tunlet receives a corresponding event record that contains the parameters of the function (i.e. mode and value). For example the tunlet receives a notification when a task explicitly changes the fragment size by calling `pvm_setopt(PvmFragSize, size)`. Moreover, this event record will also determine if a task changes the communication mode (indirect, direct).

To be able to calculate the optimal value of the message fragment size, the tunlet must receive appropriate information about a number of messages and their sizes. The tunlet then requires the instrumentation of communication calls like `pvm_send()`, `pvm_recv()`, `pvm_mcast()`. For receiving the records of each event it registers the callback. When these records come, the tunlet preprocesses them to gather statistics about messages: how many messages of a given size were sent.

The tunlet checks the conditions that must be passed before the optimal size is calculated. If the application uses indirect mode, then the tunlet activates the communication mode tuning (as described in Section 6.5). The appropriate tuning action is invoked, and from now on the application will be executed in direct mode. Additional condition is the number of possible changes. We have experimentally set this value to 3, because more changes resulted in out of memory errors. If the fragment size has been already tuned more than 3 times, the whole technique is deactivated and the inserted instrumentation is removed. When event records are received, the tunlet actualizes its statistics. If a task exchanges a significant number of messages ($NumMessages > NumMsg_{min}$) and their total size exceeds $Size_{Min}$, then the calculation of the optimal value is triggered.

Currently the tunlet implements the formula presented above. Basing on the gathered statistics, the tunlet calculates the average message size and its standard deviation. The tuning action is activated only when some conditions are fulfilled. In our implementation the difference between current and optimal values must reach Diff_{min} and the communication must be significant during the analyzed period (Comm_{min}, e.g. 20%).

The tuning action includes one-time function invocation `pvm_setopt(PvmFragSize, OptimalFragSize)` that changes the current fragment size. The invocation must be synchronized with the application execution inserting a breakpoint into the entry of the function `pvm_send()`. It is done in the same way as described in Section 6.5.4 for PVM communication mode.

6.7.5. Experiment 1

To **compare the application performance for different PVM message fragment sizes**, we have prepared a synthetic, PVM master-worker program basing on the same program as in the case of comparing both the communication and encoding performance (see Section 6.5.5 and 6.6.5). This program exchanges messages in the same manner as in the experiments described above. We executed it for message sizes ranging from 4KBytes to 4096KBytes. At startup, the master task configures the PVM message fragment size and later whole application execution is performed with this size. We have executed the experiment for various message fragment sizes – from 4KBytes to 512KBytes. All tests were conducted in the same environment as in the case of communication and encoding mode (homogeneous and dedicated, aows6, aows7).

Measurements and results

All experiments were conducted using the direct communication mode. Table 6.9 proves that the default fragment size is the best choice for small message sizes (less than 4KB). We can observe that the results from applying fragments sizes bigger than 4KB vary slightly (-1.5% to +3.45%). This can be explained by experimental error rather than real benefits. The more significant improvements can be noticed when the data size reaches 4096. This is because the physical message size exceed 4096 (data size + header and other data) and it does not fit into the 4KB buffer.

FragSize [B]	4096	16386		65536		262144		1048576	
MsgSize [B]	Time[ms]	Time[ms]	Difference	Time[ms]	Difference	Time[ms]	Difference	Time[ms]	Difference
4	0,56	0,56	0,00%	0,56	0,00%	0,56	0,00%	0,57	-1,79%
8	0,58	0,56	0,00%	0,58	0,00%	0,56	3,45%	0,56	3,45%
16	0,6	0,59	1,67%	0,59	1,67%	0,58	3,33%	0,58	3,33%
32	0,6	0,59	1,67%	0,61	-1,67%	0,6	0,00%	0,61	-1,67%
64	0,64	0,62	3,13%	0,63	1,56%	0,61	4,69%	0,63	1,56%
128	0,67	0,68	-1,49%	0,67	0,00%	0,67	0,00%	0,68	-1,49%
256	0,78	0,75	3,85%	0,75	3,85%	0,73	6,41%	0,78	0,00%
512	0,91	0,9	1,10%	0,9	1,10%	0,9	1,10%	0,92	-1,10%
1024	1,25	1,23	1,60%	1,24	0,80%	1,23	1,60%	1,25	0,00%
2048	1,81	1,78	1,66%	1,78	1,66%	1,79	1,10%	1,79	1,10%
4096	4,8	2,54	47,08%	2,56	46,67%	2,53	47,29%	2,55	46,88%

Table 6.9. Detailed measurements gathered by changing the message fragment size in a round trip application for small messages.

We concentrated then on the changing message fragment size for messages bigger than 4KB. Figure 6.12 presents the benefits gathered by changing the fragment size in our ping-pong tested application. For a clarity of the figure, we show only sample fragment sizes. Detailed measurements of conducted experiments are shown in Table 6.10. The benefits detailed in the column "Benefits [%]" for a given fragment size are related to the profits gained from changing the default fragment size to a corresponding one.

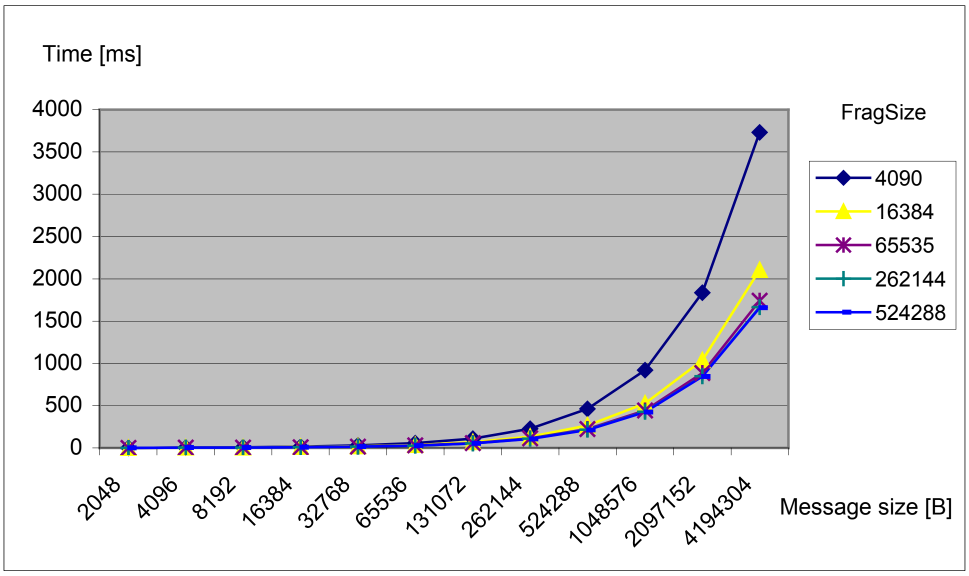


Fig. 6.12. Benefits gathered by changing the message fragment size in a ping-pong application.

We can observe that for bigger messages, the benefits of using larger fragment sizes are significant (up to 55%). The performance improvement is generally very notable starting from 8KB fragment size (e.g. for 4KB message we gain about 45% changing the fragment size from 4KB to 8KB). The bigger the fragment size is, the bigger improvement we obtain. However, the difference of changing the fragment sizes between other than default sizes is not so drastic. For example, for 4KB message, we obtain high profits by changing from default fragment size to any bigger one. However, once changed, we do not see significant differences between 8KB, 16KB, 32KB, and bigger fragment sizes. If a message fits the fragment, then no additional operations must be done, it is just sent to the receiver. So, there is no real need for a bigger fragment. In general the problem occurs in the case when a fragment size is smaller than a message size.

FragSize [B]	4090	8192		16384		32768		65535	
MsgSize [B]	Time [ms]	Time [ms]	Benefits [%]	Time [ms]	Benefits [%]	Time [ms]	Benefits [%]	Time [ms]	Benefits [%]
2048	1,77	1,73	2,28	1,75	1,11	1,77	0,07	1,77	-0,22
4096	4,54	2,50	45,02	2,51	44,68	2,50	44,84	2,51	44,68
8192	7,99	6,20	22,43	4,13	48,37	4,12	48,47	4,14	48,16
16384	15,35	11,13	27,52	9,57	37,63	7,42	51,70	7,47	51,32
32768	28,93	21,03	27,30	17,85	38,30	16,20	44,00	14,03	51,53
65536	55,64	41,80	24,88	34,11	38,70	31,22	43,88	29,47	47,03
131072	110,14	81,31	26,18	67,19	39,00	61,05	44,57	57,54	47,76
262144	226,45	162,22	28,36	133,08	41,23	119,86	47,07	113,31	49,96
524288	460,66	323,96	29,67	264,40	42,60	237,86	48,37	224,41	51,28
1048576	920,62	649,58	29,44	525,61	42,91	472,33	48,69	443,56	51,82
2097152	1833,94	1348,81	26,45	1036,70	43,47	939,90	48,75	884,42	51,77
4194304	3730,53	2617,17	29,84	2105,77	43,55	1860,23	50,13	1739,84	53,36

FragSize [B]	4090	131072		262144		524288	
MsgSize [B]	Time [ms]	Time [ms]	Benefits [%]	Time [ms]	Benefits [%]	Time [ms]	Benefits [%]
2048	1,77	1,75	0,77	1,76	0,34	1,79	-1,45
4096	4,54	2,50	44,86	2,51	44,80	2,50	44,88
8192	7,99	4,26	46,76	4,25	46,85	4,12	48,45
16384	15,35	7,47	51,32	7,47	51,32	7,45	51,47
32768	28,93	14,03	51,51	13,95	51,79	13,95	51,78
65536	55,64	27,20	51,12	27,08	51,33	26,85	51,74
131072	110,14	55,84	49,30	53,40	51,52	52,74	52,11
262144	226,45	109,70	51,56	107,99	52,31	104,93	53,66
524288	460,66	217,17	52,86	214,75	53,38	212,99	53,76
1048576	920,62	432,53	53,02	426,94	53,62	425,02	53,83
2097152	1833,94	858,37	53,20	846,72	53,83	844,00	53,98
4194304	3730,53	1688,32	54,74	1666,26	55,33	1657,24	55,58

Table 6.10. Benefits gathered by changing the message fragment size in a round trip application.

It is also notable that the point where the benefit can increment quickly is the point of equality of the message size and fragment size. Considering for example 4KB and 8KB message, we can observe the following phenomena: if the fragment size is set from 4KB to 8KB, we get 45% of benefits for 4KB messages, but only 22% for 8KB messages. This situation is caused by the necessity for allocation of additional buffers. 4KB message fills the 8KB buffer, but 8KB message does not. Internally, PVM puts additional information into the buffer, and hence it must prepare the second buffer to put there the rest of the 8KB sending message.

Here we must point out that one can have a sensation that the best way is to put the biggest fragment size and each problem will be solved. If it was set in this way, the memory resources can be quickly exhausted. Therefore, a good idea is to find a balance between available memory and the optimal fragment size. In some cases, it will be more adequate to apply smaller fragment size and not use the entire memory, as the benefits from the bigger fragment size can be insignificant. For example, the difference between 128KB and 512KB fragment size for 4MB message reaches only 0,8%.

6.7.6. Experiment 2

In order to **compare the performance of applying tuning of the PVM message fragment size in a real application**, we conducted experiments using the **Integer Sort (IS) kernel benchmark** – the same application that was tested for PVM communication and encoding mode tuning. All its executions were performed in the same as above environment (homogeneous and dedicated using 4 machines: aows1, aows6, aows7, aows8).

Measurements and results

Table 6.11 presents the results of the IS Kernel Benchmark experiments in two different tuning scenarios. In the first scenario, the application was executed without any tuning under standard PVM 3.4. The second test was performed under PVM, but MATE tuned the message fragment size. In this scenario, the tunlet requested the appropriate instrumentation in all of the tasks to examine the conditions as well as number and sizes of the messages. One of the first actions was to change the communication mode. By default the application used indirect mode and since all required conditions were accomplished (the experiments were conducted in a small NOW environment), the tunlet decided to apply the direct mode. In continuation, the analysis indicated that the default fragment size was improper, because each of the tasks sent the series of very small messages (4B and

16B) as well as large messages (over 1MB). The requested tuning action increased the fragment size. After the change, the application remained stable and no further tuning was performed. The benefits are significant – we obtained about 28% better application performance.

In this case the intrusion reached 4,9%. It is bigger that in the previous tests (for communication and encoding mode tuning). It resulted from further inserted instrumentation (more functions were instrumented and more tuning actions were performed) and a higher volume of collected measurements (more instrumentation means more measurements sent for analysis). Despite the intrusion, the introduced changes produced the significant results improving the total execution time of 28%.

No.	Tuning scenario	Execution time [sec]	Tuning benefits [sec]	Intrusion [sec]
1.	PVM (no tuning)	732	-	-
4.	PVM + message fragment size tuning indirect	769	-37 (-5,1%)	27 (3,5%)
3.	PVM + message fragment size tuning direct	523	209 (28,5%)	26 (4,9%)

Table 6.11. The measurements of PVM communication performance when applying dynamic tuning of the message fragment size in the NAS IS application.

6.7.7. Conclusions

This tuning technique tries to minimize the internal PVM message buffers fragmentation thus improving the message sending times and in consequence the communication time. We have proved that selecting the more optimal message fragment size significantly reduced the execution time of communication intensive application. The drawback of this technique results from the necessity to calculate the statistics of sizes of sending messages what implies higher monitoring intrusion. The technique is considered dynamic since it adapts the application to changes in its behavior selecting the optimal message fragment size depending on sizes of exchanged message.

6.8. Merging PVM tuning techniques

We wanted to **compare the PVM performance when applying different tuning techniques at the same time**. To achieve this goal, we focused on the PVM tuning techniques and tuned PVM communication mode, PVM encoding mode and PVM message fragment size. All techniques were applied simultaneously during one application execution. We conducted the experiments on NAS Integer Sort kernel benchmark (described above). All tests were conducted in the same environment as in the case of all PVM tuning techniques (homogeneous and dedicated, aows1, aows6, aows7, aows8).

Measurements and results

Table 6.12 presents the results of merging different PVM tuning techniques and applying them all together on the IS Kernel benchmark. For comparison, we also placed here previously described measurements obtained by performing different techniques separately on the IS benchmark.

In the first scenario, the application was executed in original version under standard PVM 3.4 without any tuning and was used as a reference result. The other tests were performed under PVM and the MATE environment. The second scenario references the Section 6.5.6 (a 17,5% benefit in execution time), the third scenario Section 6.6.6 (any tuning actions were performed as originally the application used the data-raw encoding mode) and the fourth scenario to Section 6.7.6.

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	203 (27,7%)	28 (~5,3%)

Table 6.12. Comparison of the measurements of PVM performance when applying different tuning techniques in the NAS IS application.

Finally, in the fifth scenario, we conducted all the described tuning scenarios in the same execution. Both communication mode tuning and fragment size tuning was applied successfully. The total intrusion inserted into the application execution was higher in comparison to the rest of the scenarios. It reached about 5,3% and resulted from further inserted instrumentation and a higher volume of collected measurements. **Despite the intrusion, the introduced changes produced the best results, improving the total execution time up to 27,7%.**

6.9. Workload balancing

This tuning technique intents to balance the amount of work that is distributed by the master task to each worker task taking into account capacities and load of the machines where the application is running.

6.9.1. Motivation

Dynamic load balancing is a technique which aims to distribute work among the processes to avoid some processes being idle while others only wait for work and do nothing. Load imbalance is caused by two factors:

- heterogeneous computing and communication powers
- varying amount of distributed work

A very important aspect of efficiency is idle time within the processes. The best situation would be to have all processes busy doing useful work during the application execution. However, slower or overloaded machines and/or incorrect work distribution may significantly increase the idle time of processes and influence into the application execution time. Our goal is to balance and distribute correctly the work among the available processes taking into account capacities and load of the machines the application runs on.

Typically, in the master/worker paradigm, a master process in each iteration distributes the work among worker processes and waits for their response. When the master receives results from the workers, it may distribute the work again. There are many cases the master must synchronize all the results from all the workers before the next work distribution. This situation might be especially inefficient if we consider heterogeneous environment. It causes the following problem: synchronization may affect significantly the execution time if there are slower machines because all the processes must wait idle for the slowest ones. If the work is distributed in segments that are too large, then the processes on slower machines need more time to manage such an amount of data and the rest of the processes wait too long doing nothing for the next work distribution. On the other hand, theoretically, ignoring the communication overhead, we can minimize the idle time if we distribute the work in the smallest possible units. However, in many network environments latency and bandwidth might have a significant influence into the unit transfer time. Therefore, if the work is distributed in too small segments, slave processes might wait idle for data when the master is occupied sending a big amount of work units. Such a scenario will suffer from high communication overheads.

There are many well known algorithms available such as Trapezoid Self Scheduling (TSS) [Tze93], Guided Self Scheduling (GSS) [Pol87], Factoring Scheduling [Hum92], and many more. For example, one of the most simple solution is fixed chunking of the work to be processed. Each chunk (tuple) contains the same amount of work measured by data items. The work size can be calculated using the formula:

$$\frac{N}{\sum_{i=1}^{P} P_i}$$

where N is the total number of data items to be computed, Pi is the estimated processing power and P is the number of processes. For example, if N=1000, Pi=1 (all processors of equal power), P=10, the optimal tuple size is 100. This formula gives the optimal behavior only in the case of specific conditions: homogeneous machines where one process is executed on one processor.

For our studies and experimental work we chose the Factoring Scheduling algorithm. The work is divided according to a factor into a set of different-size chunks called tuples. Obviously, different program input can significantly change the work distribution using the same tuple size calculation algorithm. The presence of the real factor ($0<f<=1$) results then in a better adaptation to both input and environment changes (machine load, network load).

The workload balancing goal therefore is to minimize the idle time and calculate the right amount of work for each process. In this research the tuning technique considers an algorithm to calculate and assign the optimal amount of work for each process considering efficiency of machines. Load balancing should be achieved because the fast computers will automatically process more amount of work than the slower ones. Moreover, an optimal work distribution may also depend on dynamic factors such as input data, network load and so on. Before the application execution, developers do not know these parameters, hence they cannot distribute the work properly. Therefore, it can be beneficial to dynamically tune the work size by adapting it to changing conditions.

6.9.2. Applicability and conditions

To apply this technique in practice, we assume the following requirements. An application must be written using M/W paradigm. The application is iteration-based, namely the same operations done by the tasks are performed repeatedly in a number of iterations. During one iteration the master process distributes the work to all worker processes and then waits for the results. It must synchronize the results before the next iteration. Data being distributed is independent, namely one must be able to divide data in a separate independent set of work units. Moreover, calculation time cannot depend on the data content, it may depend only on the data size. Finally, worker processes cannot exchange data between themselves to calculate and provide results.

The application processes the work basing on the scheme of many iterations. To apply workload balancing the iterations must present two main characteristics: the number of iterations should be significant and each iteration should last a certain time. From the one hand, there cannot be only few iterations in the whole application execution because in such a case it would not be possible to see benefits gained from changing the workload. This technique is feasible and efficient for problems that appear many times during the execution. The load balancing may be also time consuming and hence the tuning actions might be performed after a certain time has passed.

The tuning of workload is supposed to be applied before the iteration starts. So, if there is a small number of iterations and all are performed, none improved can be reached. Therefore, more iterations are, better the load balancing may be. From the other hand, one iteration should not over passed some time thresholds. To calculate the optimal factor, the network load and machine load are taken into consideration. If one iteration is very large, then the tuning can be no so efficient since the environment can differ. This issue is more flexible than the number of iterations. If there are many iterations during the application execution, each iteration can last more time. The maximal possible iteration time can be adjustable. Moreover, the algorithm that decides to apply tuning should also consider the prediction of the environment changes.

To use this tuning technique the application must be prepared. This is performed by introducing necessary source code changes and recompiling the program. This technique enters into the set of techniques that are called cooperative. Therefore, although the tunlet provides all the necessary functionality, the application must be adapted and aware of the possible changes recommended by the tunlet. For example, when modifying the factor value, the application must be aware of this change and apply in the next iteration a new modified value for work distribution.

6.9.3. Solution

In our factoring approach, the work is partitioned according to a factor into a set of different-size tuples. One tuple is distributed by the master task to one worker task. If a worker task has finished the tuple process and is free, then it receives the next tuple. This cycle is repeated till all the tuples of the work are processed. In this algorithm the work divided into tuples is distributed to workers according to the workers' demand.

In this tuning technique we had to consider distinct issues such as:

- How to calculate work tuples (factoring algorithm)
- How to distribute the work according to the given factor
- How to calculate the optimal factor and what is required to calculate this value.

In the next paragraphs we explain how these issues were solved.

Assuming there are P parallel workers, a threshold T>0 (minimal tuple size) and a factoring value (0<f<=1), the **sizes of factored work tuples** are calculated according to the following algorithm [L44]:

```
R0 = N (initial work size)
Repeat
      For each P
            Gi = Ri * f / P         // Gi: ith tuple size
            Ri+1 = Ri - (P*Gi)      // Ri: remaining work size
until Ri < T.
```

The example tuple sizes calculated according to the presented above factoring algorithm are shown in Table 6.13.

Work size (N)	Number of workers (P)	Factor (f)	Threshold (T)	Tuples
1000	2	1	1	500,500
1000	2	0.5	1	250,250,125,125,63,63,32,32,16,16,8,8,4,4,2,2,1,1
1000	2	0.5	16	250, 250, 125, 125, 62, 62, 32, 32, 16, 16, 16, 16
1000	2	0.7	1	350, 350, 105, 105, 31, 31, 10, 10, 3, 3, 1, 1
1000	4	1	1	250, 250, 250, 250
1000	4	0.5	1	125, 125, 125, 125, 62, 62, 62, 62, 32, 32, 32, 32, 16, 16, 16, 16, 8, 8, 8, 8, 4, 4, 4, 4, 2, 2, 2, 2, 1, 1, 1, 1
1000	4	0.5	16	125, 125, 125, 125, 62, 62, 62, 62, 32, 32, 32, 32, 16, 16, 16, 16, 16, 16, 16, 16
1000	4	0.7	1	175, 175, 175, 175, 52, 52, 52, 52, 16, 16, 16, 16, 5, 5, 5, 5, 1, 1, 1, 1, 1, 1, 1, 1,

Table 6.13. Examples of tuple sizes for different factors.

As we have mentioned in the applicability section, the application must be prepared for the tuning actions and hence it must implement adequately the **work distribution**. For example, the algorithm of the work distribution can be written as follows:

```
For each iteration
      //according to the factoring algorithm
      Calculate work tuples for a given factor
      //first work distribution to all workers
      For each worker
            Send corresponding tuple
      //if there are still tuples to be processed
      While processed tuples < calc number of tuples
            Receive results
            Send corresponding tuple
```

Next algorithm that is used in this tuning technique calculates the **optimal factor value**. The pseudo-code for this algorithm is presented below:

```
For each processor
      Calculate Latency
      Calculate Bandwidth
      Calculate work unit execution time (i.e. processor speed)
For each factor f (0<f<=1)
      Calculate tuples
      Calculate application iteration execution time
```

```
                While there is work left
                      For each process (using time counter)
                            If process is free
                                  Assign tuple for process
                      Time++
      Chose the minimal time of application iteration
      Set factor value to the minimal factor value
```

The presented algorithm simulates the execution of the program iteration for each possible value of factor f. The simulation uses basic measurements such as network latency, bandwidth and current speed for each processor. The algorithm simulates the complete iteration by assigning subsequent work tuples to next free processor. It takes into account the time for sending the work, time for processing the work on the selected processor in function of its current speed (i.e. capacity and current load), and time for returning the result. The iteration finishes when all tuples has been processed, and the simulation records its execution time. The algorithm, after performing iteration for all possible factors, returns the value f for the iteration with the shortest execution time.

The Figure 6.13 presents a result of an example execution of the simulator for factors f ranging from 0 to 1 in scenario with TotalWork = 1000, four processors, one 3 times slower than the others.

Considering the algorithm presented above the tuning technique must receive a set of metrics to calculate the optimal factor. These metrics are:

- Network bandwidth
- Network latency
- Average worker speed expressed as the work unit processing time

We can monitor the PVM functions responsible for exchanging messages, e.g. `pvm_send()` and `pvm_recv()` during run-time and this enables us to calculate the necessary metrics.

During program execution, the simulation algorithm estimates the optimal values for each iteration. If the optimal factor value differs significantly from the current value, the tuning procedure is invoked. The tuning action changes the factor by updating the variable value in the master process. Once it has been modified, the application can recalculate the tuples in the next iteration.

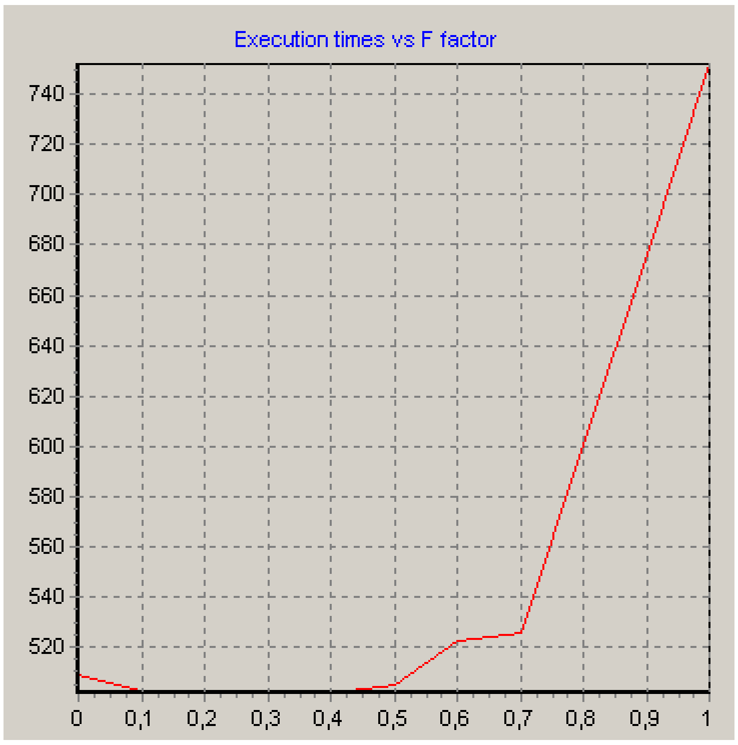


Fig. 6.13. Simulated iteration time in function of factor f (N=1000, P=4, T=1, relative speeds: 1,3,3,3).

6.9.4. Implementation

The tunlet implementing this technique must check the configuration of the virtual machine by handling notifications related to tasks and hosts – callbacks `Application::SetTaskHandler`, `Application::SetHostHandler` (see Chapter 5).

During the execution, we can monitor the PVM functions responsible for exchanging messages, e.g. `pvm_send()` and `pvm_recv()`. In particular, by monitoring: send entry/exit, receive entry/exit events in the master process, and receive entry/exit and send entry/exit in all worker processes, we are able to perform all necessary measurements. This is illustrated on Figure 6.14.

To simplify the implementation, we have assumed network latency to be 1ms. Network bandwidth can be computed using the following formulas:

$T_{comm} = T_{latency} + n * T_{bandwith}$

$T_{bandwith} = (T_{comm} - T_{latency}) / n$

T_{comm} = receive exit time on master - send entry time on master – tuple processing time on a worker

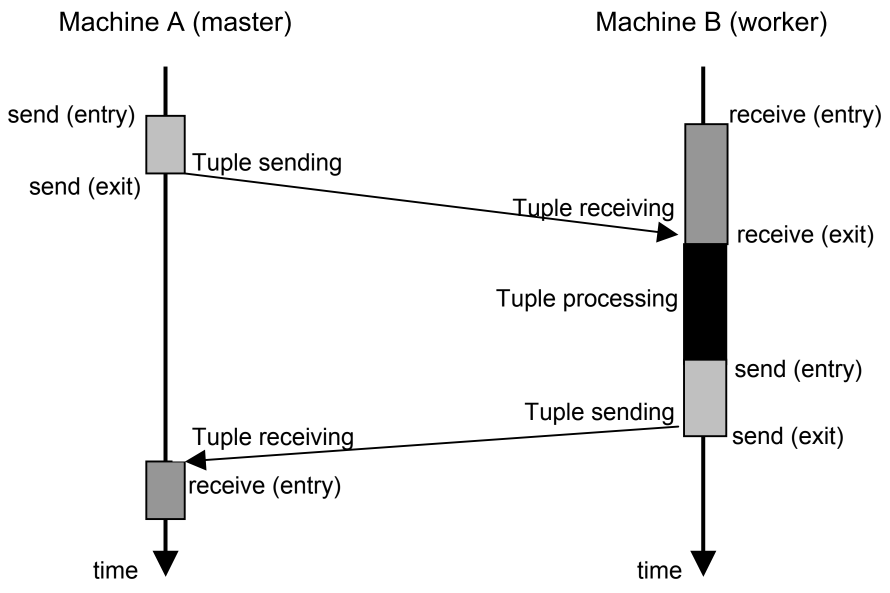


Fig. 6.14. Tuple processing model.

Work unit processing time is computed as the averaged value of the tuple processing time divided by number of work units per tuple. This is calculated individually for each worker. The average is taken for all work units processed during last iteration. This assumes the total iteration time is not very large and the average can express current machine load. If the iteration time is long, this should be changed to an average taken for a given time window. Each iteration updates the work unit processing time value for each worker and this way we can keep track of dynamic variations in the load of the machines. It must be pointed out that this approach is suitable only if the work unit processing time depends on the work unit size and not on its content.

To find the optimal value of the factor the performance model must provide the prediction of the iteration time. For this purpose we implemented the simulation of the application iteration time according to the algorithm presented in the previous section.

If a new value of the factor is different than the current one, then the tuning action should be performed. The tuning action includes the modification of the value of specific predefined application variable. This variable represents the factor of the work and its name is well known to the tunlet. When a next iteration is performed in the application, first, the new value of the factor should be applied to recalculate the work tuples, and then the work can be performed. The variable modification does not need to be synchronized, as the application will use its value next time the iteration of work processing starts.

6.9.5. Experiment 1

The goal of this experiment was to **investigate the workload balancing profitability**. To perform the experimental work we have developed a synthetic master-worker application based on the requirements presented in Section 6.9.2. In each iteration, the master task calculates the tuples according to the factor and then sends a determined amount of work to each worker. When a worker receives a tuple, it processes data and sends the changed data back to the master. The master task waits for results from any worker and when it receives them, it checks if there are tuples left. If it is the case, it sends the next tuple to the worker. In the opposite way, it waits for all results from the rest of the workers and having all tuples processed it puts them together and goes to the next iteration. By default the factor had a value 1, what means that the total work is divided into the same number of tuples as the number of workers. We executed the program with 60 iterations where each iteration processed the total work of 10.000 integers.

We have conducted our experiments in three scenarios:

- **Homogeneous and dedicated machines** – fast machines and no external load – in this scenario, we have used homogeneous machines (aows1, aows6, aows7, aows8) which built a dedicated environment (i.e. COW cluster). This scenario is the same as in the case of the previous experiments done with all the PVM tuning techniques (e.g. Section 6.5.5).
- **Heterogeneous and dedicated machines** – one machine slower and no external load – in this scenario, we have used heterogeneous machines (aows1, aows6, aows7,aows10) which built a dedicated COW cluster as has been described in the previous point. The Aows10 machine is slower in comparison to other workstations what was indicated in Table 6.1
- **Heterogeneous and non-dedicated** – one machine slower and external load – in this scenario, we based on the heterogeneous environment (aows1, aows6, aows7,aows10). Moreover, we have introduced a controlled, synthetic “external” load. For this purpose we used our Load Generator tool. As mentioned before, we generated 50% CPU load

on a given node for the duration of 20 minutes. In this way, we simulated the real conditions with multiple users working in the cluster (i.e. NOW cluster).

Measurements and results

We present the comparison of execution times of the synthetic application in Figure 6.15 and detailed measurements in Table 6.14. We show the times obtained for all the three scenarios in which the application was executed. For each scenario the synthetic application was executed without and with tuning. We can see that we gathered benefits in each scenario by balancing the workload.

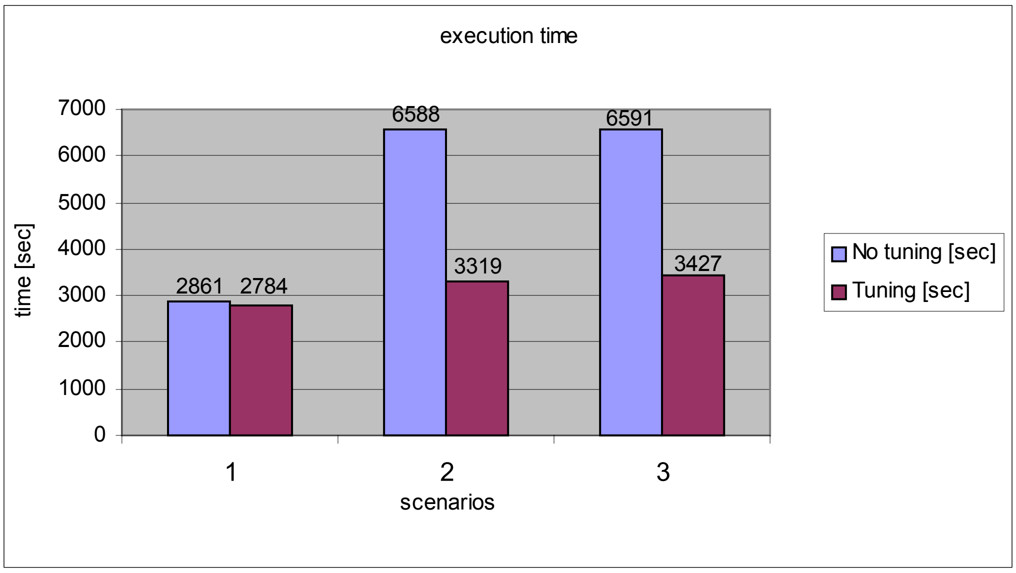


Fig. 6.15 Comparison of execution times of the synthetic application in different scenarios.

No.	Scenario	Original execution [min]	Tuned execution [min]	Intrusion [sec]	Intrusion [%]	Benefit [sec]	Benefit [%]
1.	Homogeneous, dedicated	47,7	46,4	64,8	2,33	77,4	2,71
2.	Heterogeneous, dedicated	109,8	55,3	64,8	1,95	3269	49,63
3.	Heterogeneous, non-dedicated	109,9	57,1	64,6	1,89	3164	48,01

Table 6.14. Detailed measurements of workload balancing of the synthetic application in different scenarios.

In the first scenario, although the application was executed in the homogeneous and dedicated environment, the tuning was profitable. The benefits were small (~2,7%), but we see that the workload balancing made the application execution faster. This situation was caused by the application configuration. We set the application to create 4 workers. As it was executed on 4 homogeneous machines, each machine run 1 worker, but one of the machines run also master processes (in total 2 processes). In this sense, the initial work distribution were not ideally balanced, and thus the tuning actions helped to improve the performance at least in a small percent.

The second scenario presents the highest benefits obtained. The application was executed in the heterogeneous environment where one machine was significantly slower than others. We can notice that original application lasted more than 56% (more than an hour) longer than in the homogeneous environment. Applying the workload balancing in the second scenario we gained ~49%. This high improvement was reached because of the heterogeneity of the machines. The faster worker processes had to wait for the slowest one to receive the next part of the work. Applying workload factoring, the work was divided into tuple, and hence it could be performed more rapid by the faster workers.

In the third scenario we obtained more or less the same execution times as in the second scenario for both cases: when executing original application, and when applying tuning. In the second scenario we had one slow machine (aows10), in this scenario we had the same slow machine (aows10), but we run additional external load on the fast machine (aows6). The similarity of the execution times in comparison to the second scenario results in the influence of the external load. The external load was not so significant that the slowest machine speed. Faster machine together with the loaded machine performed all the work faster than the slowest machine. In other words, it required more time to process the requested tuples than these 3 machines with one externally loaded.

Intrusion time is already included in the time given for tuned execution. In general the overhead caused by MATE is small – about 2% of the improved execution time. The intrusion is slightly different in three scenarios. Such a situation is mainly caused by different number of tuning actions. If the factor must be changed more frequently then more tuning actions are performed and hence the intrusion is higher.

6.9.6. Experiment 2

The goal of the next experiment was to **compare the performance of a real application applying workload balancing**. To conduct our experiments, we selected a computation-intensive parallel program called Xfire. **Forest Fire Propagation application (Xfire)**

[Jor98] was developed at Universitat Autonoma de Barcelona [L45]. The Xfire application is a PVM based implementation of the simulation of the fireline propagation. It calculates the next position of the fireline considering the current fireline position and different aspects as weather (wind, temperature, moisture), vegetation and topography (terrain). It was developed for use in any network of workstations.

There are several models in the literature to describe the behavior of forest fire and studies the movement of the fireline. The Xfire application simulates the fireline propagation basing on the Andre-Viegas model [And94]. The Xfire defines the fireline as a set of sections where each section contains a set of points. A section must be desegregated to calculate the individual progress of each point for a time step. When the progress of all the points have been calculated, it is necessary to aggregate the new positions of the points to rebuild the fireline. To simulate the fireline propagation, Xfire divides the fire spread into two models: global and local. The global model allows for the partitioning of the fireline into sections and for the aggregation of sections into next fireline position applying numerical algorithm. While aggregating and calculating a new fireline position, the fireline can expand and hence in certain circumstances new points can be added. It must be pointed out that the fireline sections are independent, but the end-points of each section are shared with its neighboring sections. The local model calculates the movement of each individual point. While evaluating a point, it uses numerical algorithms and takes into account static and dynamic conditions (i.e. wind, vegetation, topography) defined as numerical model.

The fireline propagation process can be summarized in the following steps:

1. Subdivision of the fireline $\phi(t)$ into a partition of sections $\delta_i\phi(P_i,t)$, with length $\Delta s_i.(\Delta s = max\{\Delta s_i\})$. In this step the model specificity is in the order (0,1,2) of the sections adopted, and the process of subdivision.
2. Resolution of a certain Local Problem for each section $\delta_i\phi(P_i,t)$, giving as result a particular virtual fireline $\Phi_{v,i}(\Delta t)$. The specificity is in the local problem formulated posed.
3. Aggregation and coupling of the information inherent to the set $\{\Phi_{v,i}(\Delta t\}\}$, finally were resulting in the definition of $\Phi(t+\Delta t)$. The specificity is the coupling principle postulated for group the movement description of the set of sections.

The forest fire propagation simulation involves different steps that requires complex calculations, and hence the fire spread computation can be then time-consuming. First implementation of the Xfire project was sequential and run on the PC. However, due to poor performance, the developers of the Xfire decided to implement it in the parallel way. They utilized data parallelism, i.e. the calculation of the movement of each section (local model) is done in parallel. The fireline is desegregated into N sections and each section is performed by distinct processes distributed among the resources of the parallel machine. It can be done in this way, since the model considers that the sections are independent.

Xfire is the PVM-based application and it follows a master-worker paradigm. A master process generates a partition of the fireline and distributes it to the workers. A worker process calculates the local fireline propagation. The general algorithm of this application using the master-worker paradigm is the following:

```
Master process:
    1. Get the initial fireline
    2. Generate a partition of the fireline (sections) 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 process:
    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 neighbors).
    3. Return the new section to the master.
```

All tests with the Xfire application were conducted in the same three scenarios as we described in the Section 6.9.5 for the experiments with the synthetic application.

Support tools

For purpose of the tests, we have designed and implemented two additional tools that allowed us to perform the experiments:

- Load Monitor – distributed cluster monitoring tool
- Load Generator – external load generator tool

The first tool, Load Monitor, was developed in order to provide detailed load measurements of all machines in the cluster. It was implemented as a PVM-based program using master-worker paradigm. The program is implemented as a set of distributed sensors running on each workstation and one centralized monitor that collected and synchronized the data. Load Monitor consists of two tasks, namely MasterMon and SlaveMon. The master task controls the SlaveMon creation (starts/stops the slave task on all nodes of the virtual machine). The slave task monitors the node and saves information to a file. Using this tool we were able to obtained detailed information about CPU and process statistics like CPU idle, kernel, user, wait times, memory consumption, paging statistics, and so on.

The second tool, Load Generator, was created in order to simulate external load conditions. Load generator was implemented as a simply C++ program. The main program controlled the generation of external load in the cluster by starting/stopping the computation for a selected period of time. The program, given a load function, was able to generate the appropriate usage of CPU. This was done by repeatedly executing the computational loop and the sleep statement with the millisecond intervals. For example, for constant load function Load(t)=60% the program occupied 60% of the CPU time during whole execution.

For all our tests with external load, on a selected node we have configured the tool to generate Load(t)=50% load during 20 minutes with time start delay of 20 minutes. In result, the generator waited for 20 minutes sleeping and then generated the 50% CPU load for 20 minutes. We used this configuration to simulate a real example of multi-user workload.

Measurements and results

Figure 6.16 presents the comparison of execution times of the Xfire application in different scenarios. We can see that in each scenario we gathered benefits by adapting the workload to the changing conditions. Detailed measurements of conducted experiments are shown in Table 6.15.

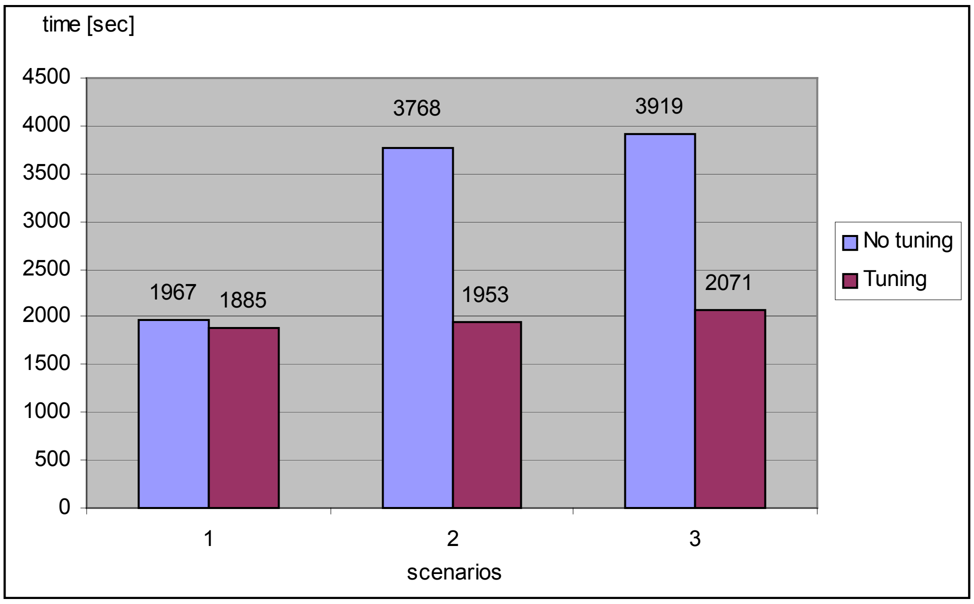


Fig. 6.16 Comparison of execution times of the synthetic application in different scenarios.

No.	Scenario	Original execution [sec]	Tuned execution [sec]	Benefit [sec]	Benefit [%]
1.	Homogeneous, dedicated	1967	1885	82	4,17
2.	Heterogeneous, dedicated	3768	1953	1815	48,17
3.	Heterogeneous, non-dedicated	3919	2071	1848	47,15

Table 6.15. Detailed measurements of workload balancing of the synthetic application in different scenarios.

All presented results have the same tendency as in the case of the results obtained from the experiments conducted with the synthetic application. As they were performed in the same scenarios and the application had similar behavior to the previous synthetic application, the reasoning of the benefits reached here is similar (see Section 6.9.5). In the first scenario, the profitability was relatively small (~4,17%), but the tuning was worth to apply. In the second and third scenario we improved the performance by nearly 50%.

6.9.7. Conclusions

Looking at these experiments we can notice that MATE adapted well the applications to the existing changing conditions. The work factoring and work load balancing gave us promising benefits. Although it can be applied only to the specific small-range applications that must accomplish a set of requirements and be prepared for changes, the profits reached are significant. Moreover, the developer might not know in what kind of environment the application will run and hence it is very desirable to use MATE and adapt dynamically the application to the changing conditions.

Obviously, the implementation of the factoring algorithm presented here can be improved. We distinguished many possibilities, as factoring of the work that remains to perform, better calculations of the latency and bandwidth to predict more adequately the network behavior or adaptation of the threshold T that represents the minimal tuple size. However, though simple factoring implementation, it gave us really good profits.

6.10. Number of workers

This tuning technique intents to optimize the number of workers assigned to perform a specified amount of work in the master/worker application.

6.10.1. Motivation

When considering efficiency and idle times, another point to tune besides the workload balancing is the number of worker processes in master/worker application. If there are too many workers, they can be idle waiting for the data from the master. On the other hand, when there are insufficient worker processes, the master process becomes idle waiting for results. Therefore, a very important issue is the adequate number of workers in order to minimize the execution time while maintaining the requested efficiency.

6.10.2. Applicability and conditions

This technique is applicable for applications written using M/W paradigm. The application structure is similar as it was in the case of workload balancing tuning technique. The application is based on the iterations where the tasks perform repeatedly all operations. During each iteration the master distributes the work to a specified number of workers and then waits for the results. It must synchronize the results before the next iteration. Data being distributed must be independent one from another, i.e. one must be able to divide data in a separate independent set of work units. In addition, data processing time cannot depend on the data content, but only on the data size. Finally, worker processes cannot exchange data between themselves to calculate and provide results.

The condition of the iteration-based application structure implies the existence of the significant number of iterations. As workload balancing, this tuning technique is beneficial when the operations are done in many iterations. If there is a small number of repetitions, the tuning overhead might be high and the improvement might not be seen.

The drawback of this technique is the consumption of resources. If there are new workers to be spawned, the new machines (processors) are required for them. There is no sense to run a new worker on the same machine where another worker is already running. In such a situation we will not gain anything since the CPU time is divided between both workers.

Similarly to the workload balancing, number of worker problem belongs to the cooperative tuning techniques. The application must be prepared by the developer for the potential changes. In general the application must contain the special variable that represents the number of workers. During execution, the application should be aware of the current number of workers and if it is different from the previous one, the new number must be applied. If there are workers to be added, the adequate number of workers should be spawned, otherwise redundant workers should be deleted. Such addition/deletion can be done only between two iterations because it is hard to change the current work distribution already being processed. Once the number of worker has been adjusted, the work can be distributed adequately to all running workers.

6.10.3. Solution

In this solution [Ces03] we will use the following terminology:

- tl, λ. Network parameters (time overhead per message and inverse bandwidth)
- V, $v_{m,}$ v_i. Size of task sent to worker i in bytes (v_i). Size of results sent back to master in bytes (v_m), and total data volume ($V=\Sigma_j (v_i + v_m)$).
- n = current number of workers in the application.
- Tt, Tc, tc_i. Time that each worker spends to process a task (tc_i), total computing time ($Tc=\Sigma_i tc_i$), and total execution time (Tt).
- Nopt = number of workers needed to obtain the minimum Tt (best performance).

To model the behavior of master/worker applications, we can use the performance model described below:

- At the beginning, the master sends one task to each worker, the time spent for this operation is

  `n*tl (network overhead) + λ*v`$_i$ `(last task communication time).`
- Next, we must add the processing time of one worker, `tc`$_i$ `(Tc/n).`
- In order to evaluate what happens to results sent back to the master we only need to count the communication time for the last message, which is `tl + λ*v`$_m$.
- The total iteration time is formed by adding these quantities together, and gives:

  ```
  Tt = n*tl + λ*vi + Tc/n + tl + λ*vm,
  as λ*vm + λ*vi = λ*V/n,
  ```
- We obtain:

  $Tt = (tl*n^2 + \lambda*V + 2*Tc)/n + tl$ **(1)**
- If we calculate `dTt/dn = 0` for expression (1) we obtain an expression to calculate the number of workers needed to minimize `Tt`, which is:

  `Nopt = sqr( (λ*V + Tc) / tl )` **(2)**

It can be seen that the *measure points* needed for this performance model are:

- tl and λ, which can be calculated at the beginning of the execution.
- Tasks and results size (v_i, v_m), which can be captured by monitoring communication functions (`pvm_send`, `pvm_recv`)
- The time workers spend on each task to calculate the total computing time (`Tc`).

The performance functions that have to be evaluated are:

- Expression (1) – to predict the application performance for any number of workers
- Expression (2) – because it indicates what has to be done to obtain the best application performance under the conditions that hold for a certain time period.

In order to optimize the number of workers, it is necessary to monitor during run-time the PVM functions responsible for exchanging messages, e.g. `pvm_send()` and `pvm_recv()`. In particular, by monitoring: send entry/exit, receive entry/exit events in the master process, and receive entry/exit and send entry/exit in all worker processes, we are able to perform all necessary measurements. This is similar to solution presented in section 6.9.4.

The presented performance model is evaluated after each iteration. If the computed optimal number of workers differs from the current value, the associated tuning procedure is invoked. If the number of workers should be changed, the solution recommends the master to add new or remove existing worker/workers.

6.10.4. Conclusions

This tuning technique adapts the number of workers assigned to perform a specified amount of work to changing environment conditions. It requires the application to be prepared for the possible changes, i.e. adding or removal of worker tasks. Moreover, the environment where the application executes is required to provide new machines when they are necessary. When these conditions are met, the technique is able to estimate the application performance by means of the analytical model, calculate and apply the optimal number of workers.

Chapter 7
Conclusions and future work

Parallel, distributed and large-scale grid programming offers high computing capabilities to the users in many scientific research fields. The performance of applications written for such an environment is one of the crucial issues. The main goal of parallel and distributed computing then is to obtain the highest performance of the application. Such applications can be useless and inappropriate if their performance is poor and under acceptable limits. Developers of parallel applications are responsible of providing their best possible behaviors but face up to many problems that must be solved. If such applications are to fulfill their promises, this implies the need for the systematical testing, analysis and optimization of their behavior. However, these tasks are very complicated when performing without any automation and especially for non-expert programmers.

It is necessary then to provide good, reliable and easy tools that automatically carry out tasks of the performance analysis of parallel programs and their behavior optimization. Such tools could help programmers to improve the performance providing them with appropriate and sufficient information about the application behavior.

Therefore a new idea has arisen. The very profitable solution is to provide a developer with automatic real-time tuning of a program. A developer is relieved then from duties of program behavior analysis and optimization, as well as from intervention into a source code. Instead of manual changes of a source code, an automatic tuning of a parallel program is performing during run-time. Such approach does not require a developer intervention nor access to the source code of the application. The running parallel application is automatically monitored, analyzed and tuned on the fly without need to re-compile, re-link and restart it.

7.1. Conclusions

The main objective of this thesis was to show that the performance of distributed parallel programs can be improved automatically during run-time. Our goal was to investigate this idea and prove that in general it works, is applicable, effective and useful. We also wanted to demonstrate that it is possible to support a user with a concrete functioning environment for automatic dynamic tuning. This thesis had to provide the good basis on how to solve optimization problems of parallel programs. A conclusion of this thesis is that although the dynamic tuning is complicated and hard task, not only it is possible, but gives real improvements of application performance. This methodology appears as a powerful technique to accomplish the successful performance of applications with dynamic behavior.

We started our work by researching well-known approaches and techniques for the application performance measurement, analysis and optimizations. Very important task was to find out what exactly is in the developer's hands when using each of these methods. We reviewed example tools that correspond to different performance measurement approaches. In order to see how is the performance analysis and optimization problem solved by others, we analyzed the most popular tools and extracted experiences from their developments. Once we knew the problem area, we could clearly distinguish that the conceptual model of the dynamic tuning consists of three main parts, namely monitoring, analysis and tuning.

Our first idea was to optimize any application without its source code. In this sense, it would be very challenging, and the work probably would be one of the most successful and useful. However, with time, we saw that due to incomplete application information this kind of tuning is extremely hard or even impossible. The performance analysis without knowledge about what the application does and dynamic modifications of unknown application structures is very complicated. It is not realistic to assume that any modification on any application in any environment can be done on the fly. We concluded then that this is a big limitation of dynamic tuning and the key question when dynamically optimizing a program is what can be tuned in it.

We distinguished different layers in the application: application-specific code, standard and custom libraries (API + code), operating system libraries (API + code), hardware. The lower the layer, the more well known information we can obtain. It means that we can extract well defined bottlenecks common for many applications and define their solutions. For such performance problems there is no need for external preparations and all can be done automatically. The upper the layer, the less information we have about the application. In this case, it is required to provide a knowledge about the specific application problems and solutions. Therefore, we differentiated two tuning approaches: automatic and cooperative. In the automatic approach the application is treated as a black-box, because no application-specific knowledge is provided by the programmer. In the cooperative approach we assume that an application is tunable and adaptable since the developer must provide application-specific information and prepare an application for the possible changes. To make the solution homogeneous for both the automatic and cooperative tuning approach, we decided that the application should be represented by a set of necessary information required for the monitoring, analysis and tuning. We defined that the application knowledge consists of measure points, performance model, tuning point/action/synchronization.

Having the conceptual model, different approaches and application knowledge defined, we focused on the requirements that have to be taken into consideration as well as what techniques we could use providing dynamic optimizations of parallel applications. The most important requirements were the parallel application control, the performance on-line analysis, and the run time monitoring and tuning. The principal technique that we could use for dynamic tuning purposes was dynamic instrumentation. By applying this method, it is possible to monitor, analyze and tune a parallel program during run-time. Moreover the application source code is not required. We have devoted big attention to the dynamic instrumentation, in particular to know the library called DynInst that supports this technique. DynInst is an API for run time code generation and permits the changes of code in a running program. We wanted to know all details about the API features and its interface provided to the user, as well as how we could use it practically for our purposes. We implemented many small examples using the DynInst API and probed its effectiveness. DynInst is a flexible and efficient platform-independent library and intrusion included into the running application is very small.

We decided to prove experimentally that dynamic tuning is effective and it is very profitable for users to take advantages of a tool that supports the automatic dynamic optimization functionalities. We devoted a big attention to design and implementation of such a tool. Our development concluded in the working environment called MATE. It includes the monitoring, analysis and modifications of the application on the fly without stopping, recompiling or rerunning the application. In this way, the MATE environment tries to adapt the application to the dynamic behavior. MATE can be applied to many performance bottlenecks that may appear during the execution of these applications.

MATE is suitable for the applications that do not have a stable behavior and change from run to run according for example to the input data or to the environment they are running (dynamic characteristics of e.g. heterogeneous non dedicated clusters). If the applications have a regular and stable behavior, it could be sufficient tuned it once. When the tuning process has been completed, the application can be manually changed and executed as many times as necessary without introducing any intrusion during the application execution.

Currently, our environment can be treated as the prototype for complete future implementation. There are still many aspects that remain for considerations and improvements. However, conducting experiments with MATE we showed that it is possible to dynamically tune applications and obtain benefits. Results of the performed tests were very promising and indicated that our prototype is applicable and effective. Obviously, all components of our environment cause intrusion and influence into the application execution, but we demonstrated that there are many examples where it is smaller than the profits gained from the performed improvements.

7.2. Future work

Many questions remain open after the work done with dynamic automatic tuning area and in particular with MATE. During our research, we encountered many opportunities for new investigation and exploration lines. We also determined possibilities for improvements of the MATE environment to make it stable, more efficient and really useful. In this section we briefly present some of these issues.

7.2.1. Global and local analysis

For the purposes of this thesis we assumed the performance analysis based on the global view of the application, that is taking into consideration all processes of the application and their interactions. Such analysis is feasible for environments with a relatively small number of nodes and serves for the inter-processes bottlenecks. The global analysis requires much information to be sent via network to the analysis component. This becomes a bottleneck if the number of nodes gets higher. If we consider problems related only to a given process without looking at other processes, the analysis can be performed locally. Scalability problem and local bottlenecks can be solved by distributing the analysis process. For example, a part of the performance analysis could be performed locally considering the locally available information, while global analysis could resolve problems caused by inter-node relationships. This approach will require certain changes, such as finding a set of tuning techniques specific for the local analysis and changes in the MATE implementation.

7.2.2. Performance analysis

In the current work the performance analysis is based on a tunlet concept. Tunlets contain the analysis logic to actually perform the dynamic tuning of the potential performance problems. Each tunlet implements code related to one concrete bottleneck that can occur in the application: how to detect it, how to overcome it and finally how to actuate inside the application. In this thesis we were focusing on investigating tuning techniques separately. However, we do not consider overall performance of the application nor how one tuning technique can influence on another one. For example, if the communication to computation ratio is very low, it might be better not to use any of the communication tuning techniques. The intrusion caused by the tuning techniques may be high and hence the application performance might be significantly decreased. If the ratio is adequate, then the intrusion although the same as in the previous case, is not so significant because we obtain high profitability. Moreover, in certain conditions MATE should take into consideration dependencies between different performance problems and associated tuning techniques.

One of the possible and interesting investigation lines for this problem would be a different approach to the run time performance analysis [Mar03]. The analysis could be based on the hierarchical model of the potential application bottlenecks. In this approach a set of bottlenecks forms a performance problem catalog. The catalog is not predefined and hard-coded, but it can be expressed in a declarative manner using e.g. APART Specification Language (ASL) [Fah00, Fah01]. ASL is a declarative specification language that uses high-level abstractions called performance properties to represent common performance problems. One bottleneck is described by one property that expresses the specific types of performance behavior in a program. Properties are based on conditions dependent upon certain performance metrics. The existence of properties is associated with some level of confidence and with severity that estimates their importance.

The performance analysis is based on the detection and evaluation of existing properties. The most severe properties represent performance problems. The analysis starts with the selection of top-level properties for all the application processes. The selected properties are first pre-evaluated in order to determine the required performance metrics. Next, the measurement collection is performed. When the performance data becomes available, the selected properties are evaluated and ranked by their severity. The most severe is expanded and its sub-properties are selected for further evaluation. The process continues the top-down search until reaching the most specific property. Once the problem is detected, the problem solution should be invoked by means of tunlets. In this sense, each property can be associated with one or a set of tuning actions. In this approach, the tool can provide its performance problem catalog and during the analysis process, must be able to interpret and evaluate the declared knowledge. This method appears as flexible and extensible so that expert users can customize or extend the catalog to their specific requirements.

7.2.3. Metrics

One of the possible bottlenecks inside the MATE environment is the event-based analysis. Although this approach is the most precise and the most flexible as events contain the detailed information about what happened, when, where and in which circumstances, it is quite invasive when a number of application processes grows rapidly. The complementary solution that could allow for minimizing the intrusion is dynamic profiling. It allows one to periodically obtain the statistical (aggregated) information about selected performance metrics and hence significantly reduce the amount of information to be transferred. The dynamic profiling differs from traditional profiling by enabling the insertion and removal of metrics dynamically during run-time. Profiling works at a function level and supports basic primitives such as timing and counting statistics. Additionally, the low-level operating system statistics could be gathered to provide constant general overview of the application performance (including metrics such as CPU-time, I/O time, memory usage).

7.2.4. Provision of the application knowledge

The MATE environment must provide the possibility to add new tuning techniques. Current version bases on tunlets that are implemented as dynamically loaded shared libraries. Such library is quite easy for incorporation into the MATE environment, but the user must provide a specific C/C++ implementation. Moreover, in this case many problems with the compiler versions may appear. First solution has been presented in Section 7.2.2 where we have been talking about the new approach to the performance analysis. The performance problem catalog can be externally provided in a declarative manner using ASL properties.

If this investigation line is not continue, other good solution would be to declare a tunlet (with all its required information, namely measure points, performance model, tuning points/action/synchronization) externally by means of e.g. XML or other custom-declarative language. The Analyzer module should then interpret such a tunlet declaration and carry the analysis on basing on the read information. It can be quite difficult, since there are many issues to consider, but such a solution would be very flexible, extensible and platform independent.

7.2.5. Tuning techniques

The presented tuning methodology is general and can be applied to improve many applications. In this work we concentrated on the C/C++ and PVM-based applications. However, there are many other programming languages as well as custom problem-specific libraries with plenty of bottlenecks. Therefore, in future we can focus on investigation of other tuning techniques to cover wider range of applications. The set of conducted experiments gave us new ideas on future tuning techniques. We present the examples of some tuning techniques considering different tuning layers:

- Application level
    - Work aggregation in pipelining applications
    - Automatic selection of the most appropriate algorithm
        - Sorting
        - Matrix calculations
        - Structures' representations – e.g. linked list vs. arrays
    - Other application-specific problems
- Custom library level
    - Numerical libraries specific problems
        - PETSc (Portable, Extensible Toolkit for Scientific Computation) [Bal97]

        - ScalaPAK [Bla97]
    - MPI-specific problems
    - PVM
        - Patterns of unicast changed for broadcast or multicast
        - Inplace data encoding mode
- Operating system level
    - I/O operations (read write)
        - Prefetch
        - Asynchronous vs. synchronous read/write operations
        - I/O buffer size

7.2.6. Instrumentation evaluation

Interesting and useful investigation line would be prediction and evaluation of the application instrumentation. MATE inserts instrumentation code for two purposes: monitoring and tuning. However, the current version of our environment does not evaluate how the inserted monitoring code and performed tuning actions influence into the global application performance. It would be very desirable to investigate the possibility of instrumentation cost prediction in order to see if performed monitoring and tuning actions were beneficial. If it was the case, the considering tuning technique would be applied. In other case it would not, since the cost could be higher than the profits obtained. DynInst library contains specific methods that allows for an estimation of the number of seconds it would take to execute the snippet. However, in our approach many snippets serve only for the invocation of the functions implemented in run time library. DynInst does not provide the estimation cost of such a function invocation. Therefore, this investigation line could be very profitable and could make our environment more efficient and more useful.

7.2.7. Event record

As a good implementation extension of MATE, we propose the inclusion of important information into the event record. Each event record should correlate performance data to source code (process, module, function). It is required to only add to event records the information about the source file and line number. This will allow a user of our environment to indicate the line in the application source code, where the particular event has occurred. We have already designed this aspect within the MATE environment, but it is still not implemented. Design as well as implementation is very complex in our case. We do not have application source code and all steps required to indicate the line number must be done analyzing the binary image of the application. The DynInst library is very helpful in performing these steps, however it still does not have the full functionality (direct, simply methods) when providing a line number of the event.

For example, in order to log event that process A has sent a message to process B, we are able to insert instrumentation into the entry of the function `pvm_send()`. It means that DynInst inserts the instrumentation code as a first operation of the function `pvm_send()` to be invoked and the rest operations are reallocated. DynInst provides methods to obtain the line number of the function body (because snippet is inserted at the entry of the function). However, we also need to log a line number of a point from where this `pvm_send()` function was called during run time. Figure 7.1 illustrates the meaning of the line number of the function body, as well as the line number of the call point. It is not sufficient provide for each event only line number of the `pvm_send()` body. This function was invoked from a specific point of the application code and to be able to relate the problem with the source code, we have to know exactly from which code point (known as call point) each event was provoked.

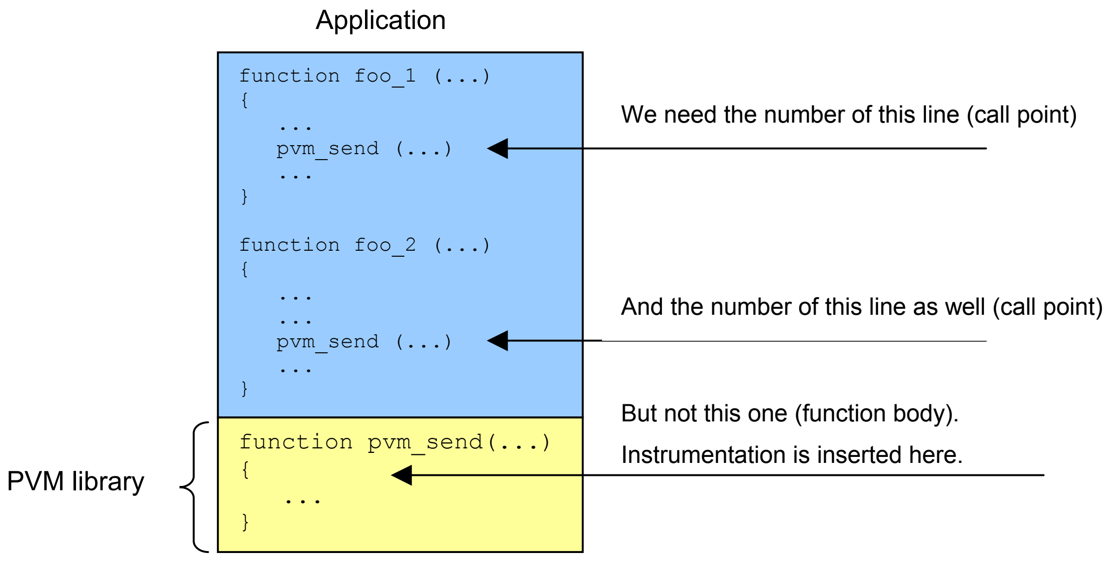

Application
function foo_1 (...)
{
...
pvm_send (...)
...
}
function foo_2 (...)
{
...
...
pvm_send (...)
...
}
function pvm_send(...)
{
...
}
PVM library
We need the number of this line (call point)
And the number of this line as well (call point)
But not this one (function body).
Instrumentation is inserted here.

Fig. 7.1. Line number of the function body and of the call point that we want to log to the event.

In order to provide the line number and source file name, from which the function was invoked, we need to scan an application image and determine for each file, for each instrumented function its call points. Then we have to be able to store all these points and log the line number of the call point together with the event record.

7.2.8. Recommendations

Once the analysis has been performed and tuning invoked, MATE, as a useful tool, should explain detected application problems. It would be very profitable to provide the user of our environment with the information about the encountered problems and their causes. Such a clear explanation of the identified problems, should indicate the most important bottlenecks and correlate them with the source code. In addition it should provide overtaken tuning actions that can be treated as potential future solutions (e.g. new optimal values of variable, settings of some important options). The whole reasoning about the detected problem could appear as the part of the tunlet. The Analyzer should simply provide the specific extensions inside the Dynamic Tuning API and each tunlet should use them to provide recommendations – that is information about the application bottleneck and how to potentially overcome it.

7.2.9. Toward grid

The rapid evolution of the Internet has brought new possibilities of connecting many clusters and parallel computers and grouping them together into a single computational platform called grid. Wide range of applications started to be developed and executed in these environments. However, developing and running programs that can draw compute power from globally distributed resources pose new challenges for the computer and computational science communities. In addition to interoperability and security issues that are the key concerns for Internet users, applications that use distributed resources as a unified compute platform must be able to achieve performance levels greater than those that could be delivered by any single resource alone [Wol02].

Performance tuning of grid application becomes even more complicated than in traditional environment due to unique grid characteristics such as time varying resource demands, heterogeneous resources, geographic distribution and network sharing. In fact, grid brings new class of challenges in performance analysis and tuning making classical approaches to performance analysis not applicable or less useful. For example post-mortem analysis that presumes repeatability, typically may not be used as the grid platform is highly dynamic and rarely repeatable. Therefore new approaches such as dynamic performance analysis and tuning must be used in order to deliver required performance.

Running the application under control of the dynamic tuning system allows for run-time adaptation of the application behavior to the changing conditions. If the target platform is changed (number of processors, processor speed, network bandwidth, etc.) the required optimizations may be different. These characteristics make the dynamic tuning approach relevant to grid systems.

Our research has focused on tuning of applications executed on cluster of workstations. However, for all the presented reasons we can see that it would be desirable to implement a new version of our MATE environment adapted to work in a grid. The new MATE version working in a grid environment appears as a very interesting research line.

Bibliography

[Ale01] A. Alexandrescu: Modern C++ Design. Generic Programming and Design Patterns Applied. Addison-Wesley. 2001.

[And94] J.C.S. Andre, 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.

[Bai94] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga: RNR Technical Report. RNR-94-007. March 1994.

[Bai95] D.H. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo, M. Yarrow: The NAS Parallel Benchmarks 2.0. Report NAS-95-020. December, 1995.

[Bal00] V. Bala, E. Duesterwald, S. Banerja: Dynamo: A Transparent Dynamic Optimization System. Hewlett-Packard Labs, PLDI. Vancouver, 2000.

[Bal97] S. Balay, W.D. Gropp, L.C. McInnes, B.F. Smith: Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. Modern Software Tools in Scientific Computing, pp. 163-202. 1997.

[Ben95] S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic, B. Chapman, M. Egg, T. Fahringer, J. Hulman, Y. Hou, E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari, V. Sipkova, B. Velkov, B. Wender, and H. Zima: Vienna Fortran Compilation System - Version 2.0 - User's Guide. October 1995.

[Ber96] F. Berman, R. Wolski: Scheduling From the Perspective of the Application. High Performance Distributed Computing 1996. Syracuse, NY, USA, August 1996.

[Bla97] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley: ScaLAPACK Users' Guide, Second Edition. SIAM, Philadelphia, PA. 1997.

[Bor03] P. Bora, C. Ribbens, S. Prabhakar, G. Swaminathan, M. Chinnusamy, A. Jeyakumar, B. Diaz-Acosta: Issues in Runtime Algorithm Selection for Grid Environments. International Workshop on Challenges of Large Applications in Distributed Environments, pp. 80-87. Seattle, Washington, June 2003.

[Buc00] B. Buck, J. K. Hollingsworth: An API for Runtime Code Patching. The International Journal of High Performance Computing Applications, 14, pp. 317-329. 2000.

[Ces02] E. César, A. Morajko, T. Margalef, J. Sorribes, E. Luque: Dynamic Performance Tuning Environment Supported by Program Specification. Scientific Programming, 10, pp. 35-44. 2002.

[Ces03] E. Cesar, J.G. Mesa, J. Sorribes, E. Luque: POETRIES: Performance Oriented Environment for Transparent Resource-Management, Implementing End-User Parallel/Distributed Applications. LNCS 2790, 141-146. 2003.

[Che00] W. Chen, S. Lerner, R. Chaiken, D. M. Gillies: Mojo: A Dynamic Optimization System. Microsoft Research. 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO). Monterey, California, December 2000.

[Cro94] M.E. Crovella, T.J. LeBlanc: The Search for Lost Cycles: A New Approach to Parallel Program Performance Evaluation. Tech. Rep. 479, University of Rochester. 1994.

[Cur76] H.J. Curnow, B.A. Wichmann: A Synthetic Benchmark. The Computer Journal Vol.19(1), 43-49. February 1976.

[Din97] P. Diniz, M. Rinard: Dynamic Feedback: An Effective Technique for Adaptive Computing. Proceedings of the ACM SIGPLAN '97 Conference on Programming Languages Design and Implementation. May 1997.

[Don88] J.J. Dongarra, J. Du Croz, S. Hammarling, R.J. Hanson: An extended set of FORTRAN Basic Linear Algebra Subprograms. ACM Trans. Math. Soft., 14, 1988 pp. 1-17. 1988.

[Esp00] A. Espinosa: Automatic Performance Analysis of Parallel Programs. Universitat Autònoma de Barcelona, Computer Science Department, Doctor Thesis. September 2000.

[Esp98] A. Espinosa, T. Margalef, E. Luque: Automatic Detection of Parallel Program Performance Problems. Lecture Notes in Computer Science, vol. 1573, pp. 365-377, Springer-Verlag. June 1998.

[Fah00] T. Fahringer, M. Gerndt, G. Riley, J. Larsson: Specification of Performance problems in MPI Programs with ASL. Proceedings of ICPP, pp. 51-58. 2000.

[Fah01] T. Fahringer, M. Gerndt, G. Riley, J.L. Traff: Knowledge Specification for Automatic Performance Analysis. Tech. report, FZJ-ZAM-IB-2001-08. 2001.

[Fah93] T. Fahringer, H.P. Zima: A Static Parameter based Performance Prediction Tool for Parallel Programs. 7th ACM International Conference on Supercomputing. Japan, July 1993.

[Gal98] J. Galarowicz, B. Mohr: Analyzing Message Passing Programs on the Cray T3E with PAT and Vampir. Proc. Of the 4th European Cray-SGI MPP Workshop. Germany, 1998.

[Gam95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing, 1995.

[Gei90]. A. Geist, T. M. Heath, B. W. Peyton, P. H. Worley: A User’s Guide to PICL: A Portable Instrumentation Communication Library. TR TM-11616, Oak Ridge National Lab. 1990.

[Gei94]. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam: PVM: Parallel Virtual Machine, A User´s Guide and Tutorial for Network Parallel Computing. MIT Press. Cambridge, MA, 1994.

[Gu94] W. Gu, J. Vetter, K. Schwan: An Annotated Bibliography of Interactive Program Steering. Technical Report GIT-CC-94-15, Georgia Institute of Technology College of Computing. 1994.

[Gu95] W. Gu, G. Eisenhauer, K. Schwan, J. Vetter: Falcon: On-line Monitoring and Steering of Parallel Programs. 5th Symposium of the Frontiers of Massively Parallel Computing. McLean, VA, 1995.

[Hea95]. M. Heath, J. Etheridge: Visualizing the Performance of Parallel Programs. IEEE Computer, vol. 28, pp. 21-28. November 1995.

[Hol03] J.K. Hollingsworth, M. Altinel: Paradyn Parallel Performance Tools, DyninstAPI Programmer's Guide. Release 4.0. University of Maryland, Computer Science Department. May 2003.

[Hol93] J. K. Hollingsworth, B. P. Miller: Dynamic Control of Performance Monitoring on Large Scale Parallel Systems. International Conference on Supercomputing. Tokyo, July 1993.

[Hol97] J. K. Hollingsworth, B. P. Miller, M. J. R. Goncalves, O. Naim, Z. Xu, L. Zheng: MDL: A Language and Compiler for Dynamic Program Instrumentation. International Conference on Parallel Architecture and Compilation Techniques. San Francisco, November, 1997.

[Hum92] S.F. Hummel, E. Schonberg, L.E. Flynn: Factoring – A Method for Scheduling Parallel Loops. CACM, Vol., 35, No.8, 90-101. August 1992.

[IEE85] IEEE Computer Society (1985), IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std 754-1985.

[Jor98] 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, November 1998.

[Kal93] L.V. Kale and S. Krishnan: Charm++ : A portable concurrent object oriented system based on C++. In Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications. September 1993.

[Kri96] S. Krishnan, L. V. Kale: Automating Parallel Runtime Optimizations Using Post-Mortem Analysis. International Conference on Supercomputing, pp. 221-228. 1996.

[Mai95]. E. Maillet: TAPE/PVM an Efficient Performance Monitor for PVM Applications – User Guide. LMC-IMAG, Grenoble, France, June 1995.

[Mar03] T. Margalef, J. Jorba, O. Morajko, A. Morajko, E. Luque: Different approaches to automatic performance analysis of distributed applications. Performance Analysis and Grid Computing, Kluwer Academic Publishers, pp.3-20. 2003.

[Mil95] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingswoth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, T. Newhall: The Paradyn Parallel Performance Measurement Tool. IEEE Computer vol. 28. pp. 37-46. November 1995.

[MPI94] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. International Journal of Supercomputer Applications, 8(3/4): 165-414. 1994.

[Nag96]. W. Nagel, A. Arnold, M. Weber, H. Hoppe: VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer 1 pp. 69-80. 1996.

[Par03] Paradyn Project: Paradyn Parallel Performance Tools, User's Guide, Release 4.0. University of Wisconsin, Computer Science Department. May 2003.

[Par95] S.G. Parker, C.R. Johnson: SCIRun: A Scientific Programming Environment for Computational Steering. SC'95. San Diego, USA, December 1995.

[Par98] J. Parecisa Viladrich: El problema del temps en la monitorizació distribuïda. MsC Thesis. Universitat Autònoma de Barcelona, Computer Science Department, June 1998.

[Pas98] D. M. Pase: Dynamic ProbeClass Library (DPCL): Tutorial and Reference Guide version 0.1. IBM Corporation, 1998.

[Pha99] C.Pham, C. Albrecht: Tuning Message Aggregation On High Performance Clusters For Efficient Parallel Simulations. Parallel Processing Letters, Vol. 9, No. 4 (1999) 521-532. 1999.

[Pol87] C. Polychronopoulos, D. Kuck: Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers. IEEE Transactions on Computers, C-36, 12, 1425-1439. December 1987.

[Rab97] R. Rabenseifner: The Controlled Logical Clock – a Global Time for Trace Based Software Monitoring of Parallel Applications in Workstation Clusters. Parallel and Distributed Processing, London, 1997.

[Ree93]. 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.

[Ree93]. D. A. Reed: Performance Instrumentation Techniques for Parallel Systems. University of Illinois, Department of Computer Science. 1993.

[Ree96] D.A. Reed, C.L. Elford, T.M. Madhyastha, E. Smirni, S.E. Lamm: The Next Frontier: Interactive and Closed Loop Performance Steering. ICPP Workshop on Challenges for Parallel Processing. August 1996.

[Rib98] R.L. Ribrel, J.S. Vetter, H. Simitci, D.A. Reed: Autopilot: Adaptive Control of Distributed Applications. High Performance Distributed Computing 1998, pp. 172-179. Chicago, August 1998.

[Rob98] J. Robb: BPatch Interface Reference version 0.1. IBM Corporation, 1998.

[Sch94] D.C. Schmidt: The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Components for Developing Client/Server Applications. Technical Report, 11th and 12th Sun Users Group. 1994.

[Sub96] K.R. Subramaniam, S.C. Kothari, D.E. Heller: A Communication Library Using Active Messages to Improve Performance of PVM. Journal of Parallel Distributed Computing, 39(2): 146-152. 1996.

[Tam99] A. Tamches, B.P. Miller: Using Dynamic Kernel Instrumentation for Kernel and Application Tuning. International Journal of High-Performance Computing Applications 13(3): 263-276. 1999.

[Tap02] C. Tapus, I-H. Chung, J.K. Hollingsworth: Active Harmony: Towards Automated Performance Tuning. SC'02. November 2002.

[Tze93] T.H. Tzen, L.M. Ni: Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. IEEE Transactions on Parallel and Distributed Systems, pp. 87-98. 1993.

[Vli95] J.M. Vlissides, J.O. Coplien, N.L. Kerth, J. Coplien: Pattern Languages of Program Design. Addison-Wesley, Reading. MA, 1995.

[Wol02] R. Wolski: Computational Grids: Current Trends in Performance-oriented Distributed Computing. Society for Industrial and Applied Mathematics. 2002.

[Yan96] J. Yan, S. Sarukhai: Analyzing Parallel Program Performance Using Normalized Performance Indices and Trace Transformation Techniques. Parallel Computing, vol. 22, pp. 1215-1237. 1996.

Referenced web links:

[L1] Portable Timing Routines (PTR) http://www.ptools.org//projects/ptr/

[L2] Parallel Tools Consortium (Ptools) http://www.ptools.org/

[L3] Silicon Graphics www.cray.com

[L4] Cray Research www.sgi.com

[L5] Xprofiler
http://publib16.boulder.ibm.com/doc_link/en_US/a_doc_lib/aixbman/prftools/xprofiler.ht
m

[L6] IBM www.ibm.com

[L7] Hardware performance monitoring http://www.npaci.edu/online/v3.6/SCAN2.html

[L8] PAPI http://icl.cs.utk.edu/papi/

[L9] PCL http://www.fz-juelich.de/zam/PCL/

[L10] http://www.fz-juelich.de

[L11] PICL tool http://www.epm.ornl.gov/picl/picl2.html

[L12] Oak Ridge National Laboratory http://www.ornl.gov/

[L13] Paragraph tool http://www.csar.uiuc.edu/software/paragraph/

[L14] Vampir tool http://www.pallas.de/pages/vampir.htm

[L15] Pallas company http://www.pallas.de

[L16] Pablo http://www-pablo.cs.uiuc.edu/Project/Pablo/ScalPerfToolsOverview.htm

[L17] Dimemas http://www.cepba.upc.es/dimemas/

[L18] CEPBA http://www.cepba.upc.es/

[L19] AIMS tool http://www.nas.nasa.gov/Groups/Tools/Projects/AIMS

[L20] NASA Ames Research Center http://www.arc.nasa.gov

[L21] Paradyn tool http://www.paradyn.org

[L22] University of Wisconsin – Computer Science Department http://www.cs.wisc.edu

[L23] ScaLAPCK http://www.netlib.org/scalapack

[L24] BLAS http://www.netlib.org/blas/

[L25] PETSc http://www.mcs.anl.gov/petsc

[L26] OCI http://www.ociweb.com/

[L27] Java HotSpot http://java.sun.com/

[L28] Parallel pattern http://www.cise.ufl.edu/research/ParallelPatterns/

[L29] DynInst library http://www.dyninst.org

[L30] University of Maryland – Computer Science Department http://www.cs.umd.edu

[L31] Abstract Syntax Tree http://www.cs.kent.ac.uk/projects/ofa/java-threads/3.html

[L32] IBM – hardware descriptions
www-1.ibm.com/servers/eserver/pseries/library

[L33] FALCON http://www.cc.gatech.edu/systems/projects/FALCON/

[L34] MOSS http://www.cc.gatech.edu/systems/projects/MOSS/

[L35] SCIRun http://www.sci.utah.edu/

[L36] SCI Software http://software.sci.utah.edu/

[L37] Autopilot http://www-pablo.cs.uiuc.edu/Software/Autopilot/autopilot.htm

[L38] Active Harmony http://www.dyninst.org/harmony/

[L39] TCP/IP tuning

http://www.nlanr.net/NLANRPackets/v2.1/autotcpwindowtuning.html

[L40] NAS www.nas.nasa.gov

[L41] Bucket sort http://www.brpreiss.com/books/opus5/html/page512.html

[L42] Gaussian distribution http://davidmlane.com/hyperstat/normal_distribution.html

[L43] XDR encoding http://www.faqs.org/rfcs/rfc1832.html

[L44] Factoring algorithm http://spartan.cis.temple.edu/synergy/user_manual.htm

[L45] Universitat Autónoma de Barcelona www.uab.es