Search of Performance Inefficiencies in Message Passing Applications with KappaPI 2 Tool

Josep Jorba[1], Tomas Margalef[2], and Emilio Luque[2]

[1] Universitat Oberta de Catalunya (UOC),
Estudis d'Informatica i Multimedia.
Barcelona, Spain,
jjorbae@uoc.edu
[2] Universitat Autonoma de Barcelona (UAB),
Computer Architecture and Operating Systems Department,
08193 Bellaterra, Spain,
tomas.margalef, emilio.luque@uab.es

Abstract. Performance is a crucial issue of parallel/distributed applications. One kind of useful tools, in this context, are the automatic performance analysis tools, that help developers in some of the phases of the performance tuning process. KappaPI 2 is an automatic performance tool, with open knowledge about typical inefficiencies in message passing applications, and it is able to detect and analyze these inefficiencies, and then make suggestions to the developer about how to improve their application behavior.

1 Introduction

Designers and developers of parallel/distributed applications, expect that their applications reach high performance indexes to meet the expectations of HPC. In this context, performance analysis is a crucial issue.

However, there is a lack of useful tools, and the most popular approach to carry out the performance analysis is the use of visualization tools [1, 2] to show several indexes obtained from the execution of the application. The analysis of these views is a difficult and time consuming task that requires a high degree of expertise from the developer.

To overcome this situation, more user-friendly tools are needed. Such tools should provide a step ahead from the visualization technics. To fullfill these requeriments, some automatic performance analysis tools have been developed, like Scalea [4] and Expert [3]. These tools take data from the execution of the application, in form of profiling or tracing data, and try to detect performance bootlnecks in the application. To identify these bottlenecks these tools uses certain performance property specification, for example, derived from the APART specification Language (ASL) [5].

KappaPI 2 is an automatic performance analysis tool that extends these ideas, enhancing the use of current knowledge of performance inefficiencies in the message passing environments, and providing support in the performance analysis process in form of recommendations directed to the developer. The application developer can use the recommendations provided by the tool to improve the performance of the applications.

2 KappaPI 2

In KappaPI 2 [8] the main goal is to provide useful hints to the application developer, to enhance the performance of their applications. The tool uses a series of specifications or performance knowledge as input data. This knowledge represents a set of parallel performance bottlenecks that can be found in the execution of parallel/distributed applications on message passing environments (MPI and PVM in our case).

The main goal of the tool architecture is to provide the mechanisms to support an automatic performance analysis tool that has the following features:

- Performance knowledge specification: Independent specification mechanisms to introduce new performance inefficiencies.
- Independence from background message passing system: The tool builds abstract entities that are independent from the particular trace file format or the message passing primitives.
- The performance inefficiency detection engine must read the performance knowledge specification and classify the inefficiens found in the trace.
- Relate inefficiencies to the source code of the application: a set of quick parsers to search for dependences in the source code must be included to determine why the inefficiency appears.

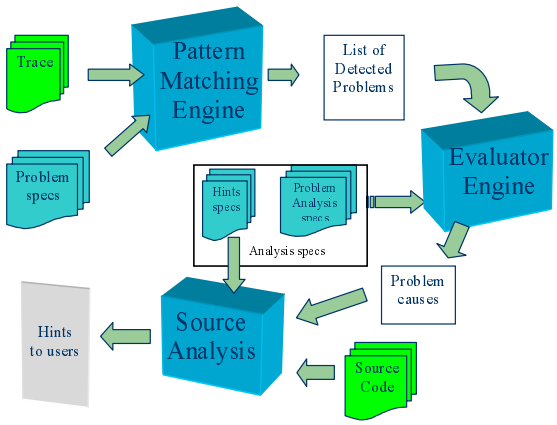

Fig. 1. Module architecture of the KappaPI 2

In KappaPI 2 (figure 1), the first step is to execute the application under the control of a tracing tool (for PVM or MPI environments) that captures all the events related to the message passing primitives that occur while running the application. Our tool uses the trace and performance inefficiency knowledge base as inputs to detect the performance bottleneck patterns defined from a structural point of view. After, it sorts the found performance bottlenecks according to certain indexes. It carries out a bottleneck

cause analysis, based on the application source code analysis, and finally provides a set of recommendations to the user, indicating how to modify the source code to overcome the detected bottlenecks.

3 Related work

Several automatic performance analysis tools can be related to KappaPI 2, including the first version of KappaPI [6], Scalea [4], Expert [3].

In the first version of KappaPI, detection of performance bottlenecks focused on idle intervals affecting the largest number of processes. Processor efficiency was used to measure execution quality, and idle processor intervals represented performance inefficiencies. The knowledge about the performance inefficiencies was a closed hard coded set of bottlenecks, and no mechanisms were provided to add new bottleneck specifications. Similar limitation also affects root cause analysis.

Scalea [4] uses an interface called JavaPSL API to specify the performance properties [5], using a Java syntax in a form of classes for each problem. The user can specify new properties (by adding new Java classes) without changing the implementation of the tool's search phase.

In Expert [3, 7], the specification of the performance properties are realised using script languages based on internal APIs (for trace manipulation, and information retrieval related to the events). Expert tries to answer the question of where the application spends time. It summarizes the indexes of each problem found and accumulate their times to compare its impact to the total application execution time. Main differences between KappaPI 2 and Expert are: a) Bootleneck specification from a structural point of view, meanwhile Expert uses functional programming (in a shell script form). b) Expert specification is based on some trace API, that the user needs to know in order to specify the bottleneck property; in Kappa PI 2 the use of the trace is only internal. KappaPI 2 also offers abstraction mechanisms from trace formats and environments: we can use different tracers in different environments (PVM, MPI). c) Expert does not offer direct techniques for source code analysis, or any kind of recommendations to help the developer to improve the application. d) In Kappa PI 2 an additional level of specification is added for bottleneck analysis causes. The user can provide knowledge about bottlenecks and their analysis process.

4 KappaPI 2 operation

KappaPI 2 needs to read the knowledge about performance inefficiencies, by means of structural bottleneck specification. From these specifications, a decission tree is build to detect and classify the performance bottlenecks.

In the analysis process, KappaPI 2 has a post-mortem approach. First, it is necessary to execute the application, with a tracer tool (adapted to the environment used, MPI or PVM), that collects all the events related to the message passing primitives that occurs during the application execution. The stored trace is used as input for the detection phase, based on the matching of the bottleneck structural specification between the specified events and their presence in the application trace.

Once a bottleneck is detected, its information is captured and stored as a match in a table of inefficiencies, which is used as classification scheme based on indexes of presence and importance of the bottleneck. When the detection phase has finished, a table of main problems is provided.

These bottlenecks, with information related to source code (where the events are produced, in form: which calls, in which file and line number in source code), are analysed to determine the causes of their occurrance. In this point a second level of specification is used. It provides information about the possible cases of the bottlenecks, based on a description of preconditions to test for determining which exact case of the match of the problem have been found. These conditions require to analyse some source code, for example to determine data dependences, or information about parameters in use, or if it is feasible to make some code transformations.

The source analysis process is done by means of structural source code representation, based in a XML like representation. We can analyze some blocks of code, and determine their structure, or see information about symbols, or detect the context of a message passing call (in a loop, conditional, ...). This representation is used to get information of a particular point of source code (point of interest in case analysis). Afterwards, some quick parsers (for small dedicated tasks) detect some of the conditions of interest for the use case evaluated.

Finally, once a case is found the tool provides a recommended action to be done on the source code to overcome or eliminate the found bottlenecks.

We have carried out a series of study cases, with different kind of tests, for the validation of the architectural phases of the tool: detection, classification, cause analysis (with extra source code analysis), and final recomenadtions for the developer.

Different tests are made using standard synthetic code, some performance avalaible benchmarks, and some real scientific applications, in terms of validation of analysis phases of the tool, and final results of improvement of the application performance.

References

1. W.E. Nagel, A. Arnold, M. Weber, H.C. Hoppe, K. Solchenbach: VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer 63, vol XII, number 1, Jan. 1996.
2. L. De Rose, Y. Zhang, D.A. Reed: SvPablo: A Multilanguage Performance Analysis system. Lecture Notes in Computer Science, 1469:352-99, 1998.
3. F. Wolf, B. Mohr, J. Dongarra, S Moore: Efficient Pattern Search in Large Traces Through Succesive Refinement. Euro-Par 2004, LNCS 3149, 2004.
4. HL. Truong, T. Fahringer, G. Madsen, AD. Malony, HMoritsch, S. Shende: On using SCALEA for Performance Analysis of Distributed and Parall el Programs. Supercomputing 2001 Conference (SC2001), Denver, Colorado, USA. November 10-16,2001
5. T. Fahringer, M. Gerndt, G. Riley, J. Larsson: Specification of Performance problems in MPI Programs with ASL. Proceedings of ICPP, pp. 51-58. 2000.
6. A. Espinosa, T. Margalef, E. Luque: Automatic Performance Analysis of PVM applications. EuroPVM/MPI 2000, LNCS 1908, pp. 47-55. 2000.
7. F. Wolf, B. Mohr: Automatic Performance Analysis of MPI Applications Based on Event Traces. In EuroPar 2000, LNCS, 1900, pp123-132, 2000.
8. J. Jorba, T. Margalef, E. Luque: Automatic Performance Analysis of Message Passing Applications using the KappaPI 2 tool. EuroPVM/MPI 2005, LNCS 3666, September 2005.