Automatic Performance Analysis of MPI Applications Based on Event Traces

Felix Wolf and Bernd Mohr

Research Centre Jülich,
Central Institute for Applied Mathematics,
52425 Jülich, Germany,
{f.wolf, b.mohr}@fz-juelich.de

Abstract. This article presents a class library for detecting typical performance problems in event traces of MPI applications. The library is implemented using the powerful high-level trace analysis language EARL and is embedded in the extensible tool component EXPERT described in this paper. One essential feature of EXPERT is a flexible plug-in mechanism which allows the user to easily integrate performance problem descriptions specific to a distinct parallel application without modifying the tool component.

1 Introduction

The development of fast and scalable parallel applications is still a very complex and expensive process. The complexity of current systems involves incremental performance tuning through successive observations and code refinements. A critical step in this procedure is transforming the collected data into a useful hypothesis about inefficient program behavior. Automatically detecting and classifying performance problems would accelerate this process considerably.

The performance problems considered here are divided into two classes. The first is the class of well-known and frequently occurring bottlenecks which have been collected by the ESPRIT IV *Working Group on Automatic Performance Analysis: Resources and Tools* (APART) [4]. The second is the class of application specific bottlenecks which only can be specified by the application designers themselves.

Within the framework defined in the KOJAK project [6] (*Kit for Objective Judgement and Automatic Knowledge-based detection of bottlenecks*) at the Research Centre Jülich which is aimed at providing a generic environment for automatic performance analysis, we implemented a class library capable of identifying typical bottlenecks in event traces of MPI applications.

The class library uses the high-level trace analysis language EARL (**E**vent **A**nalysis and **R**ecognition **L**anguage) [11] as foundation and is incorporated in an extensible and modular tool architecture called EXPERT (**Ex**tensible **Per**formance **T**ool) presented in this article. To support the easy integration

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 123–132, 2000.

of application-specific bottlenecks, EXPERT provides a flexible *plug-in* mechanism which is capable of handling an arbitrary set of performance problems specified in the EARL language.

First, we summarize the EARL language together with the EARL model of an event trace in the next section. In section 3 we present the EXPERT tool architecture and its extensibility mechanism in more detail. Section 4 describes the class library for detection of typical MPI performance problems which is embedded in EXPERT. Applying the library to a real application in section 5 shows how our approach can help to understand the performance behavior of a parallel program. Section 6 discusses related work and section 7 concludes the paper.

2 EARL

In the context of the EARL language a performance bottleneck is considered as an event pattern or compound event which has to be detected in the event trace after program termination. The compound event is build from primitive events such as those associated with entering a program region or sending a message. The pattern can be specified as a script containing an appropriate search algorithm written in the EARL trace analysis language. The level of abstraction provided by EARL allows the algorithm to have a very simple structure even in case of complex event patterns.

A performance analysis script written in EARL usually takes one or more trace files as input and is then executed by the EARL interpreter. The input files are automatically mapped to the EARL event trace model, independently of the underlying trace format, thereby allowing efficient and portable random access to the events recorded in the file. Currently, EARL supports the VAMPIR [1], ALOG, and CLOG [7] trace formats.

2.1 The EARL Event Trace Model

The EARL event trace model defines the way an EARL programmer views an event trace. It describes event types and system states and how they are related. An event trace is considered as a sequence of events. The events are numbered according to their chronological position within the event trace. EARL provides four predefined event types: entering (named *enter*) and leaving (*exit*) a code region of the program, and sending (*send*) as well as receiving (*recv*) a message. In addition to these four standard event types the EARL event trace model provides a template without predefined semantics for event types that are not part of the basic model. If supported by the trace format, regions may be organized in groups (e.g. user or communication functions).

The event types share a set of typical attributes like a timestamp (*time*) and the location (*loc*) where the event happened. The event type is explicitly given as a string attribute (*type*). However, the most important attribute is the

position (*pos*) which is needed to uniquely identify an event and which is assigned according to the chronological order within the event trace.

The *enter* and *exit* event types have an additional *region* attribute specifying the name of the region entered or left. *send* and *recv* have attributes describing the destination (*dest*), source (*src*), tag (*tag*), length (*len*), and communicator (*com*) of the message.

The concepts of region instances and messages are realized by two special attributes. The *enterptr* attribute which is common to all event types points to the *enter* event that determines the region instance in which the event happened. In particular *enterptr* links two matching *enter* and *exit* events together. Apart from that, *recv* events provide an additional *sendptr* attribute to identify the corresponding *send* event.

For each position in the event trace, EARL also defines a system state which reflects the state after the event at this position took place. A system state consists of a region stack per location and a message queue. The region stack is defined as the set of *enter* events that determine the region instances in which the program executes at a given moment, and the message queue is defined as the set of *send* events of the messages sent but not yet received at that time.

2.2 The EARL Language

The core of the current EARL version is implemented as C++ classes whose interfaces are embedded in each of the three popular scripting languages Perl, Python, and Tcl. However, in the remainder of this article we refer only to the Python mapping.

The most important class is named `EventTrace` and provides a mapping of the events from a trace file to the EARL event trace model. `EventTrace` offers several operations for accessing events: The operation `event()` returns a hash value, e.g. a Python dictionary. This allows to access individual attributes by providing the attribute name as hash key. Alternatively, you can get a literal representation of an event, e.g. in order to write some events to a file.

EARL automatically calculates the state of the region stacks and the message queue for a given event. The `stack()` operation returns the stack of a specified location represented as a list containing the positions of the corresponding *enter* events. The `queue()` operation returns the message queue represented as a list containing the positions of the corresponding *send* events. If only messages with a certain source or destination are required, their locations can by specified as arguments to the `queue()` operation.

There are also several operations to access general information about the event trace, e.g. to get the number of locations used by a parallel application.

For a complete description of the EARL language we refer to [12].

3 An Extensible and Modular Tool Architecture

The EXPERT tool component for detection of performance problems in MPI applications is implemented in Python on top of EARL. It is designed according

to the specifications and terminology presented in [4]. There, an *experiment* which is represented by the performance data collected during one program run, i.e. a trace file in our case, is characterized by the occurrence of different *performance properties*. A performance property corresponds to one aspect of inefficient program behavior. The existence of a property can be checked by evaluating appropriate conditions based on the events in the trace file.

The architecture of the trace analysis tool EXPERT is mainly based on the idea of separating the performance analysis process from the definitions of the performance properties we are looking for. Performance properties are specified as Python classes which represent patterns to be matched against the event trace and which are implemented using the EARL language. Each pattern provides a *confidence* attribute indicating the confidence of the assumption made by a successful pattern match about the occurrence of a performance property. The *severity* attribute gives information about the importance of the property in relation to other properties.

All pattern classes provide a common interface to the tool. As long as these classes fulfill the contract stated by the common interface, EXPERT is able to handle an arbitrary set of patterns.

The user of EXPERT interactively selects a subset of the patterns offered by the tool by clicking the corresponding checkbuttons on the graphical user interface. Activating a pattern triggers a pattern specific configuration dialogue during which the user can set different parameters if necessary. Optionally, he can choose a program region to concentrate the analysis process only on parts of the parallel application.

The actual trace analysis performed by EXPERT follows an event driven approach. First there is some initialization for each of the selected patterns which are represented by instances of the corresponding classes. Then the tool starts a thread which walks sequentially through the trace file and for each single event invokes a callback function provided by the pattern object according to the type of the event. The callback function itself may request additional events, e.g. when it follows a link emanating from the current event which is passed as an argument, or query system state information by calling appropriate EARL commands. After the last event has been reached, EXPERT applies a wrapup operation to each pattern object which calculates result values based on the data collected during the walk through the trace. Based on these result values the severity of the pattern is computed. Furthermore, each pattern may provide individual results, e.g. concerning the execution phase of the parallel program in which a pattern match was found.

Customizing EXPERT with Plug-Ins

The signature of the operations provided by the pattern classes is defined in a common base class *Pattern*, but each derived class may provide an individual implementation.

EXPERT currently manages two sets of patterns, i.e. one set of patterns representing frequently occurring message passing bottlenecks which is described

in the next section and one set of user defined patterns which may be used to detect performance problems specific to a distinct parallel application.

If users of EXPERT want to provide their own pattern, they simply write another realization (subclass) of the *Pattern* interface (base class). Now, all they have to do is to insert the new class definition in a special file which implements a *plug-in* module. At startup time EXPERT dynamically queries the module's namespace and looks for all subclasses of *Pattern* from which it is now able to build instances without knowing the number and names of all new patterns in advance. By providing its own configuration dialogue which may be launched by invoking a configure operation on it each pattern can be seamlessly integrated into the graphical user interface.

4 Automatic Performance Analysis of MPI Programs

Most of the patterns for detection of typical MPI performance properties we implemented so far correspond to MPI related specifications from [4]. The set of patterns is split into two parts. The first part is mainly based on summary information, e.g. involving the total execution times of special MPI routines which could also be provided by a profiling tool. However, the second part involves idle times that can only be determined by comparing the chronological relation between concrete region instances in detail. This is where our approach can demonstrate its full power. A major advantage of EXPERT lies in its ability to handle both groups of performance properties in one step.

Currently, EXPERT supports the following performance properties:[1]

Communication costs: The severity of this property represents the time used for communication over all participating processes, i.e. the time spent in MPI routines except for those that perform synchronization only. The computed amount of time is returned as severity.

Synchronization costs: The time used exclusively for synchronization.

IO costs: The time spent in IO operations. It is essential for this property that all IO routines can be identified by their membership in an IO group whose name can be set as a parameter in the configuration dialogue.

Costs: The severity of this property is simply the sum of the previous three properties. Note that while the severities of the individual properties above may seem uncritical, the sum of all together may be considered as a performance problem.

Dominating communication: This property denotes the costs caused by the communication operation with maximum execution time relative to other communication operations. Besides the total execution time (severity) the name of the operation can also be requested.

[1] In [4] the severity specification is based on a scaling factor which represents the value to which the original value should be compared. In EXPERT this scaling factor is provided by the tool and is not part of the pattern specification. Currently it is the inverse of the total execution time of the program region being investigated.

Frequent communication: A program region has the property *frequent communication*, if the average execution time of communication statements lies below a user defined threshold. The severity is defined as the costs caused by those statements. Their names are also returned.

Big messages: A program region has the property *big messages*, if the average length of messages sent or received by some communication statements is greater than a user defined threshold. The severity is defined as the costs caused by those statements. Their names are also returned.

Uneven MP distribution: A region has this property, if communication statements exist where the standard deviation of the execution times with respect to single processes is greater than a user defined threshold multiplied with the mean execution time per process. The severity is defined as the costs caused by those statements. Their names are also returned.

Load imbalance at barrier: This property corresponds to the idle time caused by load imbalance at a barrier operation. The idle times are computed by comparing the execution times per process for each call of `MPI_BARRIER`. To work correctly the implementation of this property requires all processes to be involved in each call of the collective barrier operation. The severity is just the sum of all measured idle times.

Late sender: This property refers to the amount of time wasted, when a call to `MPI_RECV` is posted before the corresponding `MPI_SEND` is executed. The idle time is measured and returned as severity. We will look at this pattern in more detail later.

Late receiver: This property refers to the inverse case. A `MPI_SEND` blocks until the corresponding receive operation is called. This can happen for several reasons. Either the implementation is working in synchronous mode by default or the size of the message to be sent exceeds the available buffer space and the operation blocks until the data is transfered to the receiver. The behavior is similar to an `MPI_SSEND` waiting for message delivery. The idle time is measured and the sum of all idle times is returned as severity value.

Slow slaves: This property refers to the *master-slave* paradigm and identifies a situation where the master waits for results instead of doing useful work. It is a specialization of the *late sender* property. Here only messages sent to a distinct *master* location which can be supplied as a parameter are considered.

Overloaded master: If the slaves have to wait for new tasks or for the master to receive the results of finished tasks, this property can be observed. It is implemented as a mix of *late sender* and *late receiver* again involving a special *master* location.

Receiving messages in wrong order: This property which has been motivated by [8] deals with the problem of passing messages *out of order*. The sender is sending messages in a certain order, but the receiver is expecting the arrival in another order. The implementation locates such situations by querying the message queue each time a message is received and looking for *older* messages with the same target as the current message. Here, the severity is defined as the costs resulting from all communication operations involved in such situations.

Whereas the first four properties serve more as an indication that a performance problem exists, the latter properties reveal important information about the reason for inefficient program behavior. Note that especially the implementations of the last six properties require the detection of quite complex event patterns and therefore can benefit from the powerful services provided by the EARL interpreter.

5 Analyzing a Real Application

In order to demonstrate how the performance analysis environment presented in the previous sections can be used to gain deeper insight into performance behavior we consider a real application named *CX3D* which is used to simulate the *Czochralski* crystal growth [9], a method being applied in the silicon waver production. The simulation covers the convection processes occurring in a rotating cylindric crucible filled with liquid melt.

The convection which strongly influences the chemical and physical properties of the growing crystal is described by a system of partial differential equations. The crucible is modeled as a three dimensional cubical mesh with its round shape being expressed by cyclic border conditions. The mesh is distributed across the available processes using a two dimensional spatial decomposition. Most of the execution time is spent in a routine called `VELO` and is used to calculate the new velocity vectors. Communication is required when the computation involves mesh cells from the border of each processors' sub-domain.

The `VELO` routine has been investigated with respect to the *late sender* pattern. This pattern determines the time between the calls of two corresponding point-to-point communication operations which involves identifying the matching *send* and *recv* events. The Python class definition of the pattern is presented in Fig. 1.

Each time EXPERT encounters a *recv* event the `recv_callback()` operation is invoked on the pattern instance and a dictionary containing the *recv* event is passed as an argument. The pattern first tries to locate the *enter* event of the enclosing region instance by following the *enterptr* attribute. Then, the corresponding *send* event is determined by tracing back the *sendptr* attribute. Now, the pattern looks for the *enter* event of the region instance from which the message originated. Next, the chronological difference between the two *enter* events is computed. Since the `MPI_RECV` has to be posted earlier than the `MPI_SEND`, the `idle_time` has to be greater than zero. Last, we check whether the analyzed region instances really belong to `MPI_SEND` and `MPI_RECV` and not to e.g. `MPI_BCAST`. If all of that is true, we can add the measured idle time to the global sum `self.sum_idle_time`. The complete pattern class as contained in the EXPERT tool also computes the distribution of the losses introduced by that situation across the different processes, but this is not shown in the script example.

The execution configuration of CX3D is determined by the number of processes in each of the two decomposed dimensions. The application has been

```
class LateSender(Pattern):
    [... initialization operations ...]
    def recv_callback(self, recv):
        recv_start = self.trace.event(recv['enterptr'])
        send = self.trace.event(recv['sendptr'])
        send_start = self.trace.event(send['enterptr'])
        idle_time =  send_start['time'] - recv_start['time']
        if (idle_time > 0 and
            send_start['region'] == "MPI_SEND" and
            recv_start['region'] == "MPI_RECV"):
            self.sum_idle_time = self.sum_idle_time + idle_time
    def confidence(self):
        return 1                  # safe criterion
    def severity(self):
        return self.sum_idle_time
```

Fig. 1. Python class definition of the *late sender* pattern

executed using different configurations on a Cray T3E. The results are shown in Table 1. The third column shows the fraction (severity) of execution time spent in communication routines and the rightmost column shows the fraction (severity) of execution time lost by late sender. The results indicate that the process topology has major impact on the communication costs. This effect is to a significant extent caused by the late sender pattern.

For example, in the 8 x 1 configuration the last process is assigned only a minor portion of the total number of mesh cells since the corresponding mesh dimension length is not divisible by 8. This load imbalance is reflected in the calculated distribution of the losses introduced by the pattern (Table 2).

Table 1. Idle times in routine VELO introduced by *late sender*

#Processes	Configuration	Communication Cost	Late Sender
8	2 x 4	0.191	0.050
8	4 x 2	0.147	0.028
8	8 x 1	0.154	0.035
16	4 x 4	0.265	0.055
16	8 x 2	0.228	0.043
16	16 x 1	0.211	0.030
32	8 x 4	0.335	0.063
32	16 x 2	0.297	0.035

However, the results produced by the remaining configurations may be determined by other effects as well.

Table 2. Distribution of idle times in an 8 x 1 configuration

Process	0	1	2	3	4	5	6	7
Fraction	0.17	0.08	0.01	0.01	0.01	0.01	0.05	0.68

6 Related Work

An alternative approach to describe complex event patterns was realized by [2]. The proposed Event Definition Language (EDL) allows the definition of compound events in a declarative manner based on extended regular expressions where primitive events are clustered to higher-level events by certain formation operators. Relational expressions over the attributes of the constituent events place additional constraints on valid event sequences obtained from the regular expression. However, problems arise when trying to describe events that are associated with some kind of state.

KAPPA-PI [3] performs automatic trace analysis of PVM programs based on a set of predefined rules representing common performance problems. It also demonstrates, how modern scripting technology, i.e. Perl in this case, can be used to implement valuable tools.

The specifications from [4] on top of which the class library presented in this paper is build, serve also as foundation for a profiling based tool COSY [5]. Here the performance data is stored in a relational database and the performance properties are represented by appropriate SQL queries.

A well-known tool for automatic performance analysis is developed in the Paradyn project [10]. In contrast to our approach Paradyn uses online instrumentation. A predefined set of bottleneck hypotheses based on metrics described in a dedicated language is used to prove the occurrence of performance problems.

7 Conclusion and Future Work

In this article we demonstrated how the powerful services offered by the EARL language can be made available to the designer of a parallel application by providing a class library for the detection of typical problems affecting the performance of MPI programs.

The class library is incorporated into EXPERT, an extensible tool component which is characterized by a separation of the performance problem specifications from the actual analysis process. This separation enables EXPERT to handle an arbitrary set of performance problems.

A graphical user interface makes utilizing the class library for detection of typical MPI performance problems straightforward. In addition, a flexible plug-in mechanism allows the experienced user to easily integrate problem descriptions specific to a distinct parallel application without modifying the tool.

Whereas our first prototype realizes only a simple concept of selecting a search focus, we want to integrate a more elaborate hierarchical concept supporting stepwise refinements and experiment management in later versions.

Furthermore, we intend to support additional programming paradigms like *shared memory* and in particular hybrid models in the context of SMP cluster computing. A first step would be to extend the EARL language towards a broader set of event types and system states associated with such paradigms.

References

[1] A. Arnold, U. Detert, and W.E. Nagel. Performance Optimization of Parallel Programs: Tracing, Zooming, Understanding. In R. Winget and K. Winget, editors, *Proc. of Cray User Group Meeting*, pages 252–258, Denver, CO, March 1995.

[2] P. C. Bates. *Debugging Programs in a Distributed System Environment.* PhD thesis, University of Massachusetts, February 1886.

[3] A. Espinosa, T. Margalef, and E. Luque. Automatic Performance Evaluation of Parallel Programs. In *Proc. of the 6th Euromicro Workshop on Parallel and Disributed Pocessing(PDP'98)*, 1998.

[4] T. Fahringer, M. Gerndt, and G. Riley. Knowledge Specification for Automatic Performance Analysis. Technical report, ESPRIT IV Working Group APART, 1999.

[5] M. Gerndt and H.-G. Eßer. Specification Techniques for Automatic Performance Analysis Tools. In *Proceedings of the 5th International Workshop on High-Level Programming Models and Supportive Environments (HIPS 2000), in conjunction with IPDPS 2000*, Cancun, Mexico, May 2000.

[6] M. Gerndt, B. Mohr, M. Pantano, and F. Wolf. Performance Analysis for CRAY T3E. In IEEE Computer Society, editor, *Proc. of the 7th Euromicro Workshop on Parallel and Disributed Pocessing(PDP'99)*, pages 241–248, 1999.

[7] W. Gropp and E. Lusk. *User's Guide for MPE: Extensions for MPI Programs.* Argonne National Laboratory, 1998. http://www-unix.mcs.anl.gov/mpi/mpich/.

[8] J.K. Hollingsworth and M. Steele. Grindstone: A Test Suite for Parallel Performance Tools. Computer Science Technical Report CS-TR-3703, University of Maryland, Oktober 1996.

[9] M. Mihelcic, H. Wenzl, and H. Wingerath. Flow in Czochralski Crystal Growth Melts. Technical Report Jül-2697, Research Centre Jülich, December 1992.

[10] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvine, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[11] F. Wolf and B. Mohr. EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs. In A. Hoekstra and B. Hertzberger, editors, *Proc. of the 7th International Conference on High-Performance Computing and Networking (HPCN'99)*, pages 503–512, Amsterdam (The Netherlands), 1999.

[12] F. Wolf and B. Mohr. EARL - Language Reference. Technical Report ZAM-IB-2000-01, Research Centre Jülich, Germany, February 2000.