Finding Inefficiencies in OpenMP Applications Automatically with Periscope*

Karl Fürlinger and Michael Gerndt

Institut für Informatik,
Lehrstuhl für Rechnertechnik und Rechnerorganisation
Technische Universität München
{Karl.Fuerlinger, Michael.Gerndt}@in.tum.de

Abstract. Performance optimization of parallel programs can be a time-consuming and difficult task. Therefore, tools are desirable that help application developers by automatically locating inefficiencies. We present Periscope, a system for automated performance analysis based on the notion of performance *properties*.

We present the overall architecture of Periscope, which consists of a set of analysis agents and show how properties of OpenMP applications are detected. We describe the set of OpenMP properties we have defined so far and the data model used in the specification of these properties. Practical tests on the feasibility of our approach are performed with a number of OpenMP applications.

1 Introduction

With creating scientific parallel programs comes the question of efficiency. Does the program make optimal use of the available resources? Is the effort of parallelization paying off or is performance lost somewhere? Assisting programmers in answering such questions by automated performance analysis is important because performance analysis can be a difficult and time-consuming task.

In this paper we present Periscope, a tool for automated performance analysis of OpenMP and MPI codes and evaluate it on finding inefficiencies in OpenMP applications. Periscope detects inefficiencies by automatically searching for performance problems that are specified in terms of *performance properties*. We present the overall design and implementation of Periscope and test it on a number of OpenMP applications.

The rest of this paper is organized as follows: In Sect. 2 we describe the overall structure of Periscope and our monitoring approach for OpenMP applications. We describe how properties are specified with respect to a data model and specify the data model for OpenMP applications. Sect. 3 then describes a number of properties we have defined for OpenMP. In Sect. 4 we test our approach on a several applications from the NAS benchmark suite, in Sect. 6 we summarize and discuss ideas for future work.

* This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG) under contract GE1635/1-1.

V.N. Alexandrov et al. (Eds.): ICCS 2006, Part II, LNCS 3992, pp. 494–501, 2006.

2 Periscope

Periscope is a system for automated performance analysis on large-scale cluster-like systems. The analysis is performed by a set of *agents* distributed over the machine. The agents are arranged in a hierarchy, node-level agents form the lowest level of the hierarchy, a master-agent integrates the performance data of the whole system and connects to the front-end of the tool with is the interface to the user. Intermediate agents form a tree with the master-agent as root and the node-level agents as leaves. All components use a registry service to register themselves and to discover higher or lower level agents, see Fig. 1.

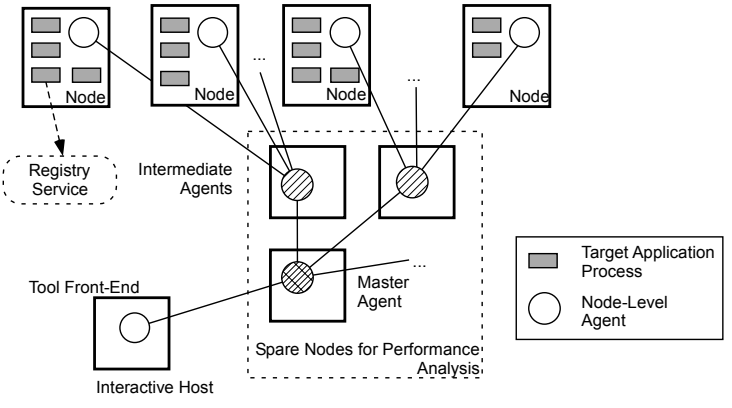

Fig. 1. The arrangement of the Periscope agents forms a hierarchy. At the lowest level, node-agents detect performance properties. Intermediate agents integrate the results of the node-level agents. A single master agent forms the connection to the tool's front-end. All components register themselves with a registry service.

While parts of the Periscope system are still under development (e.g., the higher-level agents) the functionality for performance analysis of OpenMP applications is already available in the node-level agents and this is what we evaluate in this paper. Periscope's agent hierarchy mainly serves to enable scalable analysis on large machines (hundreds of nodes), where the central collection of performance data can lead to problems related to the scalability of data management, analysis and visualization.

Periscope's node-level agents perform an automated search for performance properties of the target application. The properties are formally specified in a language called ASL (APART specification language) [1] with respect to a certain data-model that depends on the particular programming model used (message passing or shared memory programming). The data model for OpenMP is described in the next section, while the properties we have defined for OpenMP are described in Sect. 3

```
ParPerf {
  Region *reg,          // The region for which the summary is collected
  Experiment *exp,      // The experiment where this data belongs to
  int threadC,          // Number of threads that executed the region
  double execT[],       // Total execution time per thread
  double execC[],       // Total execution count per thread
  double exitBarT[],    // Time spent in the implicit exit barrier
  double singleBodyT[], // Time spent inside a single construct
  int singleBodyC[],    // Execution count in a single construct
  double enterT[],      // Time spent waiting to enter a construct
  int enterC[],         // Number of times a threads enters a construct
  double exitT[],       // Time spent to exit a construct
  int exitC[],          // Number of times a thread exits a construct
  double sectionT[],    // Time spent inside a section construct
  int sectionC[],       // Number of times a section construct is entered
  double startupT[],    // Time required to create a parallel region
  double shutdownT[],   // Time required to destroy a parallel region
}
```

Fig. 2. The `ParPerf` structure contains summary data for OpenMP constructs

2.1 Monitoring and Data Model

We monitor the execution of OpenMP applications using the POMP monitoring interface [5] and the Opari [6] source-to-source instrumenter. Opari adds calls to a POMP compliant monitoring library in and around OpenMP constructs. Periscope implements the POMP interface in a library that is linked with the target application and is thus able to observe the execution of OpenMP applications.

Our monitoring library observes the events the application generates and writes event records to a buffer that is located in a shared memory segment. The event records are processed and analyzed by a node-level agent executing on the same node as the target application. This way, we try to minimize the perturbation of the target application, as any more time-consuming analysis is performed by the node-level agent executing on a processor set-aside for performance analysis. More details about our monitoring system as outlined above can be found in [2].

The node-level analyzes the monitoring events and generates performance data corresponding to the ASL data model. For OpenMP applications, the data model is represented by the `ParPerf` data structure shown in Fig. 2.

The `ParPerf` structure holds summary (profiling) data for individual OpenMP regions (i.e., the extent of OpenMP regions and user-defined regions). The `reg` member of the `ParPerf` structure points to the region for which the data is collected. The meaning of the other members is briefly explained in the form of "comments" in Fig. 2. In [3] we describe this data model in detail. Note that not all data members are meaningful for all OpenMP regions. For example, `sectionT` and `sectionC` are only defined for a `section` construct.

3 Performance Properties

Performance properties are formally specified in ASL (APART specification language). The property specification has three major components: condition, confidence and severity. Severity describes the impact on the performance of the application that a property represents. Confidence is a measure of the certainty that a property holds and condition specifies how the property can be checked. An example ASL specification for the `ImbalanceAtBarrier` property is shown in Fig. 3.

In this example, and in general, the severity depends on the computed imbalance which is divided by a value returned by a 'ranking basis' function (RB). The ranking basis allows a scaling of the severity with respect to the experiment conducted. In all results presented in this study, the ranking basis corresponds to the overall time used by all threads (wall-clock time × number of threads). Hence the severity of a property corresponds to improvement in runtime a program would experience, if the reason for the inefficiency could be eliminated entirely.

```
property ImbalanceAtBarrier(ParPerf pd) {
  let
    imbal = pd.execT[0]+...+pd.execT[pd.threadC-1];

  condition  : (pd->reg.type==BARRIER) && (imbal > 0);
  confidence : 1.0;
  severity   : imbal / RB(pd.exp);
}
```

Fig. 3. The ASL specification of the `ImbalanceAtBarrier` property

Among others we have defined the following properties for OpenMP applications: `ImbalanceAtBarrier`, `ImbalanceInParallelSections`, `ImbalanceInParallelLoop`, `ImbalanceInParallelRegion`, `UnparallelizedInSingleRegion`, `UnparallelizedInMasterRegion`, `ImbalanceDueToNotEnoughSections`, `ImbalanceDueToUnevenSectionDistribution`, `LockContention`, and `CriticalSectionContention`.

The `ImbalanceAtBarrier` and `ImbalanceIn*` properties refer to time spent waiting at explicit or implicit barriers, respectively. For explicit (programmer-added) barriers the barrier wait time of thread n is available in the `execT[`n`]` member of the `ParPerf` structure, see Fig. 3. Implicit barriers are added to worksharing constructs and parallel regions by Opari in order to measure the load imbalance. The waiting time is available in the `exitBarT` member in this case (the barrier is added at the end of the respective construct, hence the name).

The `ImbalanceDueTo*` properties give the reason for the discovered imbalance in a more detailed fashion. `NotEnoughSections` refers to the fact that there where too few sections for the available number of threads, whereas `UnevenSectionDistribution` indicates that some threads executed more sections than

others. As an example consider a `section` construct that contains six individual `sections`. When the work per section is approximately equal and four threads execute the construct, two threads will execute 2 sections and two threads will only execute one. If, on the other hand, eight threads are used, six threads will each execute one section and the remaining two will be idle. The `sectionC` member of `ParPerf` can be used to find out the number of section constructs executed by each thread.

The `UnparallelizedIn*` properties capture the situation that time was lost due to a single thread executing a construct. For `single` constructs the severity is measured by the summed time in `exitBarT`, for `master` region, the severity is approximated by the master's time divided by the number of threads.

`CriticalSection` and `LockContention` sum up the time lost due to threads waiting to acquire a lock or to enter a critical section, respectively. The properties are based on the `enterT` and `exitT` times.

3.1 Implementation

In Periscope properties are implemented as C++ classes compiled to dynamically loadable objects (.so files). To simplify development, only the `condition` code has to be written for each property and a script generates a complete C++ class. A severity value is computed in the condition code and is returned by the `severity` method of the C++ class, the confidence is fixed to 1.0 for our prototype implementation.

Having the properties available as dynamically loadable modules allows the development and deployment of the tool separately from the "knowledge base". Without changing the main tool, new performance properties can be implemented and tested or existing ones can be modified.

4 Test Setup and Results

We tested Periscope on the OpenMP version of the NAS parallel benchmarks version 3.2. The programs in the NAS benchmark suite are derived from CFD applications, consists of five kernels (EP, MG, CG, FT, IS) and three simulated CFD applications (LU, BT, SP).

We executed the applications on a 32-CPU SGI Altix system based on Itanium-2 processors with 1.6 GHz and 6MB L3 Cache using a batch system. The number of threads was set to eight and the Periscope node-level agent was executed on a separate CPU (i.e., nine CPUs were requested for the batch runs).

The Periscope node-level agents have the ability to conduct the search for performance properties at any time during the execution of the target application (on-line performance analysis). In this study, however, we use the node-level agents in a post-mortem mode, i.e., the search for performance properties is triggered when the application finishes.

The table in Figure 4 shows all properties identified by Periscope for the NAS benchmarks. Note that for completeness this table shows all properties without applying a severity cutoff threshold. Some properties in Figure 4 have very

Property	BT	CG	EP	FT	IS	LU	MG	SP
ImbalanceAtBarrier					1	3		
ImbalanceInParallelSections								
ImbalanceInParallelLoop	12	13	1	8	2	9	12	16
ImbalanceInParallelRegion	6	9	1		2	8	2	5
UnparallelizedInSingleRegion						3		
UnparallelizedInMasterRegion	4					13	2	5
ImbalanceDueToNotEnoughSections								
ImbalanceDueToUnevenSectionDistribution								
CriticalSectionContention								1
LockContention								

Fig. 4. Performance Properties identified by Periscope. This table lists all discovered performance properties, even such with very low severity values.

Benchmark	Property	Region	Severity
BT	ImbalanceInParallelLoop	rhs.f 177--290	0.0446
BT	ImbalanceInParallelLoop	y_solve.f 40--394	0.0353
BT	ImbalanceInParallelLoop	rhs.f 299--351	0.0347
CG	ImbalanceInParallelLoop	cg.f 556--564	0.0345
CG	ImbalanceInParallelRegion	cg.f 772--795	0.0052
CG	ImbalanceInParallelRegion	cg.f 883--957	0.0038
EP	ImbalanceInParallelRegion	ep.f 170--230	0.0078
EP	ImbalanceInParallelLoop	ep.f 129--133	0.0001
FT	ImbalanceInParallelLoop	ft.f 606--625	0.0676
FT	ImbalanceInParallelLoop	ft.f 653--672	0.0304
FT	ImbalanceInParallelLoop	ft.f 227--235	0.0269
IS	ImbalanceAtBarrier	is.c 526	0.0272
IS	ImbalanceInParallelRegion	is.c 761--785	0.0087
IS	ImbalanceInParallelLoop	is.c 397--403	0.0020
LU	ImbalanceAtBarrier	ssor.f 211	0.0040
LU	ImbalanceAtBarrier	ssor.f 182	0.0032
LU	ImbalanceInParallelLoop	rhs.f 189--309	0.0011
MG	ImbalanceInParallelLoop	mg.f 608--631	0.0831
MG	ImbalanceInParallelLoop	mg.f 779--815	0.0291
MG	ImbalanceInParallelLoop	mg.f 536--559	0.0248
SP	ImbalanceInParallelLoop	x_solve.f 27--296	0.0285
SP	ImbalanceInParallelLoop	y_solve.f 27--292	0.0265
SP	ImbalanceInParallelLoop	z_solve.f 31--326	0.0239

Fig. 5. The three most severe performance properties with source-code location and severity value, identified by Periscope (only two properties were found for EP)

low severity values and do not actually represent severe performance problems. Figure 5 shows the three most severe properties identified by Periscope. All properties have severity values below nine percent. Most properties are in the range of three or four percent.

5 Related Work

A number of tools try to automate the process of performance analysis. Expert [9] performs automated search for inefficiencies in trace files. The execution of instrumented programs generates traces in the Epilog format that are analyzed by Expert. Expert presents the results in a viewer with three hierarchies, one hierarchy contains the type of inefficiency, one hierarchy shows the machine and one hierarchy shows the program's resources (files, functions,...). Recent improvements of Expert include an algebra for performing cross-experiment analysis and support for virtual topologies.

Paradyn [4] is a tool for automated on-line performance analysis based on dynamic instrumentation. In the running application Instrumentation is added and removed as required by the currently tested *hypothesis* (which performance problem exist) and *focus* (where the problem exists). An infrastructure for the efficient collection of performance data called MRNet [7] has been developed and integrated in Paradyn, while the search for performance problems is still performed centrally by Paradyn's Performance Consultant. Recently, a distributed search methodology supporting a partially distributed approach and a fully distributed approach has been added to Paradyn, called the Distributed Performance Consultant [8].

6 Summary and Future Work

Periscope is a tool for performance analysis based on the automated search for performance properties. Properties are formally specified with respect to a data-model that depends on the programming model employed. We described the data model used for OpenMP applications and a set of properties based on that data model.

We have tested Periscope and the performance properties on the OpenMP NAS benchmark suite. The study shows that Periscope is an efficient tool for the automated detection of inefficiencies. Inefficiencies were detected in each of the applications. In some applications the detected inefficiencies were not signification (e.g., EP); for MG an inefficiency of about 8 percent was discovered.

An attractive feature of Periscope lies in the fact that the specification of the tool's knowledge base is separate from the implementation of the tool itself. Performance properties are specified as C++ objects that are compiled into shared objects that are dynamically loaded by Periscope at tool startup. This allows for an easy extension of Periscope without the need to re-compile the entire tool and allows a performance expert to easily experiment with new property specifications.

Future work is planned along several directions: the data model described in this paper and the properties specification based on it only use timing data. For the future we plan to integrate hardware-counter data as well. This will enable the detection of important inefficiencies, related to cache usage for example.

References

1. Thomas Fahringer, Michael Gerndt, Bernd Mohr, Felix Wolf, Graham Riley, and Jesper Larsson Träff. Knowledge specification for automatic performance analysis. APART technical report, revised edition. Technical Report FZJ-ZAM-IB-2001-08, Forschungszentrum Jülich, 2001.
2. Karl Fürlinger and Michael Gerndt. Distributed application monitoring for clustered SMP architectures. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Proceedings of the 9th International Euro-Par Conference on Parallel Processing*, pages 127–134. Springer, August 2003.
3. Karl Fürlinger and Michael Gerndt. Performance analysis of shared-memory parallel applications using performance properties. In *Proceedings of the 2005 International Conference on High Performance Computing and Communications (HPCC-05)*, pages 595–604, September 2005. Accepted for publication.
4. Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.
5. Bernd Mohr, Allen D. Malony, Hans-Christian Hoppe, Frank Schlimbach, Grant Haab, Jay Hoeflinger, and Sanjiv Shah. A performance monitoring interface for OpenMP. In *Proceedings of the Fourth Workshop on OpenMP (EWOMP 2002)*, September 2002.
6. Bernd Mohr, Allen D. Malony, Sameer S. Shende, and Felix Wolf. Towards a performance tool interface for OpenMP: An approach based on directive rewriting. In *Proceedings of the Third Workshop on OpenMP (EWOMP'01)*, September 2001.
7. Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. MRNet: A software-based multicast/reduction network for scalable tools. In *Proceedings of the 2003 Conference on Supercomputing (SC 2003)*, November 2003.
8. Philip C. Roth and Barton P. Miller. The distributed performance consultant and the sub-graph folding algorithm: On-line automated performance diagnosis on thousands of processes. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*, March 2005. Accepted for Publication.
9. Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. In *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003)*, pages 13–22. IEEE Computer Society, February 2003.