Performance Analysis of Shared-Memory Parallel Applications using Performance Properties

Karl Fürlinger and Michael Gerndt

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

Abstract. Tuning parallel code can be a time-consuming and difficult task. We present our approach to automate the performance analysis of OpenMP applications that is based on the notion of performance properties. Properties are formally specified in the APART specification language (ASL) with respect to a specific data model. We describe a data model for summary (profiling) data of OpenMP applications and present performance properties based on this data model. We evaluate the usability of the properties on several example codes using our OpenMP profiler `ompP` to acquire the profiling data.

1 Introduction

Tuning parallel code can often be a time-consuming and complex task. Nonetheless, it is generally important for application developers that their code uses the available resources efficiently and delivers close-to-optimal performance, since improving performance is commonly the reason exploiting parallelism in the first place.

Many tools have been devised that application developers employ to analyze the performance of their code. First it is necessary to decide whether performance problems actually exist in the code. For the list of potential tuning targets it is then also necessary to identify the *reason* for the inefficiency. Knowing the reason then allows the developer to modify the code in order to remedy the performance problem.

The process of code tuning outlined above, also sometimes referred to as the measure–analyze–modify cycle [1, 13], requires the application developer to have detailed knowledge of how the application's code finally corresponds to performance delivered on some parallel machine. Various levels of abstraction added by operating system, communications middle-ware and parallel programming languages complicate the analysis of this correspondence. Furthermore, programming languages, communications middle-ware and most notably the parallel computing systems themselves evolve and change their characteristics over time and application developers need to keep track of these changes. Tradeoffs and rules-of-thumb that might be applicable for one generation of a machine can be inadequate for the next.

Performance tool builders therefore seek to support application developers by *automating* the process of performance analysis and tuning. Automation can be employed at every stage of the measure–analyze–modify cycle. While supportive tools for performance data measurement are well-understood and in widespread use (e.g., tracing and profiling tools), this is less so for approaches automating the analysis step, let alone the tuning step.

In this work we describe our advances towards automating the analysis phase of the tuning cycle, specifically for shared-memory parallel programs. Our work is based on the notion of performance *properties* that formally describe situations of inefficiency. By specifying what constitutes a performance property, a performance specialist can encapsulate domain- and platform specific knowledge that an application developer often neither has, nor wants to care about. Performance properties can also identify the reason for the inefficiency and convey hints on successful tuning strategies.

The rest of this paper is organized as follows: in Sect. 2 we give a general overview of our performance analysis approach based on performance properties. Then, in Sect. 3 we present the properties for shared-memory parallel programs and describe the data model on which the properties rely. In Sect. 4 we evaluate our approach on application examples from a test suite, designed for testing performance analysis tools (the ATS [12]) and from the OpenMP source code repository [2]. We present related work in Sect. 5 and summarize and discuss directions for future work in Sect. 6.

2 Specification of Performance Properties

Performance properties describe situations of inefficient execution. Within the APART (Automated Performance Analysis: Real Tools) working group, a language for the formal specification of performance properties was developed (the APART specification language, ASL [3, 7]).

A typical property specification is shown in Fig. 1. The specification has three parts:

- A specification of the *condition* that needs to be fulfilled in order for the property to hold.
- An expression that gives the *confidence* that the property holds.
- An expression that gives the *severity* of the property, i.e., that quantifies the impact on the performance that a particular property represents.

Condition, confidence and severity are expressed with respect to entities of a specific data model. The data model contains abstractions like regions, performance summary data structures and events for a particular programming environment (i.e., there might be a different data model for OpenMP and MPI, but a combined model for mixed-parallel code is possible as well). In Fig. 1 the data model is represented by the `SeqPerf` data structure, which contains summary

```
property MpiOvhdInSeqRegInProc (SeqPerf pD)
  {
    condition  : pD.mpiT > 0;
    confidence : 1.0;
    severity   : pD.mpiT/RB(pD.exp);
  }
```

Fig. 1. The ASL specification of the `MpiOvhdInSeqRegInProc` property.

(i.e., profiling) data of sequential (not thread-parallel) regions. The `condition` and `severity` specifications refer to the `mpiT` data member that gives the total time spent in MPI calls in a particular region. `RB()` refers to the ranking basis of the experiment, which allows the determination of the severity based on the time lost in a particular construct. In our case the ranking basis refers to the total execution time of the application. For online-performance analysis where the application execution is assumed to proceed in repeating phases, it could, however, also refer to the duration of a single phase.

ASL also contains mechanisms that allow the easy formulation of new properties based on existing ones (meta-properties and property templates). This allows for a compact and flexible specification of properties by a performance specialist. For details please refer to [3, 7].

3 Properties for Shared-Memory Parallel Programs

A number of properties have previously been devised for shared-memory parallel programs [4]. Here we take up that work and analyze what performance data can practically be derived from the execution of OpenMP programs and which properties can be based on that data.

In our work, we rely on instrumentation of OpenMP programs that utilizes the work of Mohr et al. [10]. Since OpenMP lacks a standard performance measurement interface, Mohr et al. designed such an interface (called POMP) that exposes OpenMP program execution events to performance analysis tools. In the POMP proposal functions are called when OpenMP regions are being entered or exited, for example `POMP_Parallel_fork` and `POMP_Parallel_join` are called immediately before and after a `parallel` construct.

The `POMP_*` calls are inserted by the source-to-source instrumenter Opari [11]. A performance tool implements the `POMP_*` functions and is thus able to observe the program's execution and record performance characteristics as needed. We implemented our own POMP-based profiler called `ompP` [6] to derive the necessary profiling data for the data model presented below.

```
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 (profiling) data for OpenMP constructs.

3.1 Data Model

Our ASL data model used for OpenMP shared-memory parallel programs is represented by the `ParPerf` structure holding summary data for program regions corresponding to OpenMP constructs in the target application. The `ParPerf` structure is shown in Fig. 2 in C/C++/Java-like syntax, it has the following entries

– `exp` points to a data structure that gives general information about the conducted experiment such as when the experiment started and when it ended.
– `reg` points to a `Region` structure that holds static information about the regions of the program (such as begin and end line numbers and the type of the region, e.g., `PARALLEL`, `CRITICAL`, `SINGLE`, ...).
– `threadC` gives the number of threads that executed the OpenMP construct.
– The other members of `ParPerf` hold dynamic performance data in the form of timings and counts. Not all data members are defined for all region types. For example `singleBodyT` is defined only for `SINGLE` regions. It holds the time spent *inside* a single construct. `exitBarT` is defined for parallel regions and OpenMP worksharing regions. It represents the time spent inside the implicit exit barrier added by Opari to measure the load-imbalance in these constructs[1]. `enterT` and `exitT` are only defined for `CRITICAL` regions and

[1] To measure load-imbalance in a worksharing construct, Opari adds a `nowait` clause to the construct and inserts a `barrier` at the end of the construct.

measure the time required to enter and exit the critical section, respectively. `startupT` and `shutdownT` are only meaningful for `PARALLEL` regions, these timings allow the measurement of the time lost due to thread creation and teardown for parallel regions.

3.2 Property Specification

The ASL performance properties for OpenMP code are defined with respect to the `ParPerf` data structure. We describe some of the more important properties below, other properties that have been defined but are not shown here are `AllThreadsLockContention`, `FrequentAtomic`, `InsufficientWorkInParallel`, `UnparallelizedInSingleRegion`, `UnparallelizedInMasterRegion`, `ImbalanceDueToUnevenSectionDistribution` and `LimitedParallelismInSections`.

ImbalanceAtBarrier This property refers to an explicit OpenMP `barrier` directive added by the programmer. The property measures the difference in arrival time of the individual threads at the barrier. This is usually related to a situation of load imbalance the threads encounter before arriving at the barrier. Time waited by the threads at the barrier is lost and is the basis for computing the severity.

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

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

ImbalanceInParallelRegion This property (like the very similar `ImbalanceInParallelLoop` and `ImbalanceInParallelSections`) measures imbalances of the `parallel` construct (or the respective OpenMP work-sharing constructs). Opari adds an *implicit* barrier at the end of these constructs, time spent in the this barrier is accessible via `exitBarT`.

```
property ImbalanceInParallelRegion(ParPerf pd) {
  let
     min = min(pd.exitBarT[0],...,pd.exitBarT[pd.threadC-1]);
     max = max(pd.exitBarT[0],...,pd.exitBarT[pd.threadC-1]);
     imbal = max-min;

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

CriticalSectionContention This property indicates that threads contend for a critical section. Waiting time for entering or exiting the critical section is summed-up in `enterT` and `exitT`, respectively.

```
property CriticalSectionContention {
  let
     enter = sum(pd.enterT[0],...,pd.enterT[pd.threadC-1]);
     exit  = sum(pd.exitT[0],...,pd.exitT[pd.threadC-1]);

  condition  : (pd.reg.type==CRITICAL) && ((enter+exit) > 0);
  confidence : 1.0;
  severity   : (enter+exit) / RB(pd.exp);
}
```

4 Application Examples

In this section we test our approach to automate the performance analysis of OpenMP applications based on the ATS properties defined in Sect. 3. The experiments were run on a single 4-way Itanium-2 SMP system (1.3 GHz, 3 MB third level cache and 8 GB main memory), the Intel Compiler version 8.0 was used.

4.1 APART Test Suite (ATS)

The ATS [12] is a set of test applications (MPI and OpenMP) developed within the APART working group. The framework is based on functions that generate a sequential amount of work for a process or thread and on a specification of the distribution of work among processes or threads. Building on this basis, individual programs are created that exhibit a certain pattern of inefficient behavior, for example "imbalance in parallel region".

The `ompP` [6] output in Fig. 3 is from a profiling run of the ATS program that demonstrates the "imbalance in parallel loop" performance problem. Notice the `exitBarT` column and the uneven distribution of time with respect to threads {0,1} and {2,3}.

```
R00003   LOOP pattern.omp.imbalance_in_parallel_loop.c (15--18)
 TID     execT     execC  exitBarT
  0       6.32         1      2.03
  1       6.32         1      2.02
  2       6.32         1      0.00
  3       6.32         1      0.00
  *      25.29         4      4.05
```

Fig. 3. The `ompP` profiling data for the loop region in the ATS program that demonstrates the `ImbalanceInParallelLoop` property.

This inefficiency is easily identified by checking the ImbalanceInParallel-Loop property. The imbalance in exitBarT (difference between maximum and minimum time) amounts to 2.03 seconds, the total runtime of the program was 6.33 seconds. Therefore the ImbalanceInParallelLoop property is assigned a severity of 0.32.

This example is typical for a number of load imbalance problems that are easily identified by the corresponding ASL performance properties. Other problems related to synchronization are also easily identified.

4.2 Quicksort

Towards a more real-world application example, we present an evaluation in the context of work performed by Süß and Leopold on comparing several parallel implementations of the Quicksort algorithm [16]. The code is now part of the OpenMP source code repository [2] and we have analyzed a version with a global work stack (called sort_omp_1.0 in [16]). In this version there is a single stack of work elements (sub-vectors of the vector to be sorted) that are placed on and taken form the stack by threads concurrently. Access to the stack is protected by two critical sections. The ompP output below shows the profiling data for the two critical sections.

```
R00002   CRITICAL        cpp_qsomp1.cpp (156--177)
 TID      execT     execC     enterT    enterC      exitT     exitC
  0        1.61    251780       0.87    251780       0.31    251780
  1        2.79    404056       1.54    404056       0.54    404056
  2        2.57    388107       1.38    388107       0.51    388107
  3        2.56    362630       1.39    362630       0.49    362630
  *        9.53   1406573       5.17   1406573       1.84   1406573

R00003   CRITICAL        cpp_qsomp1.cpp (211--215)
 TID      execT     execC     enterT    enterC      exitT     exitC
  0        1.60    251863       0.85    251863       0.32    251863
  1        1.57    247820       0.83    247820       0.31    247820
  2        1.55    229011       0.81    229011       0.31    229011
  3        1.56    242587       0.81    242587       0.31    242587
  *        6.27    971281       3.31    971281       1.25    971281
```

Checking for the CriticalSectionContention property immediately reveals the access to the stacks as the major source of inefficiency of the program. Threads content for the critical section, the program spends a total of 7.01 seconds entering and exiting the first and 4.45 seconds for the second section. Considering a total runtime of 61.02 seconds, this corresponds to a severity of 0.12 and 0.07, respectively.

Süß and Leopold also recognized the single global stack as the major source of overhead and implemented a second version with thread-local stacks.

Profiling data for the second version appears below. In this version the overhead with respect to critical sections is clearly smaller than the first one (enterT and `exitT` have been improved by about 25 percent) The overall summed runtime reduces to 53.44 seconds, an improvement of about 12 percent, which is in line with the results reported in [16]. While this result demonstrates a nice performance gain with relatively little effort, our analysis clearly indicates room for further improvement.

```
R00002   CRITICAL        cpp_qsomp2.cpp (175--196)
 TID      execT     execC     enterT    enterC     exitT     exitC
  0        0.67    122296       0.34    122296      0.16    122296
  1        2.47    360702       1.36    360702      0.54    360702
  2        2.41    369585       1.31    369585      0.53    369585
  3        1.68    246299       0.93    246299      0.37    246299
  *        7.23   1098882       3.94   1098882      1.61   1098882

R00003   CRITICAL        cpp_qsomp2.cpp (233--243)
 TID      execT     execC     enterT    enterC     exitT     exitC
  0        1.22    255371       0.55    255371      0.31    255371
  1        1.16    242924       0.53    242924      0.30    242924
  2        1.32    278241       0.59    278241      0.34    278241
  3        0.98    194745       0.45    194745      0.24    194745
  *        4.67    971281       2.13    971281      1.19    971281
```

5 Related Work

Several approaches for automating the process of performance analysis have been developed.

Paradyn's [9] Performance Consultant automatically searches for performance bottlenecks in a running application by using a dynamic instrumentation approach. Based on hypotheses about potential performance problems, measurement probes are inserted into the running program. Recently MRNet [15] has been developed for the efficient collection of distributed performance data. However, the search process for performance data is still centralized.

The Expert [17] performs an automated post-mortem search for patterns of inefficient program execution in event traces. As in our approach, data collection for OpenMP code is based on POMP interface. However, Expert performs tracing which often results in large data-sets and potentially long analysis time, while we only collect summary data in the form of profiles.

Aksum [8, 5], developed at the University of Vienna, is based on a source code instrumentation to capture profile-based performance data which is stored in a relational database. The data is then analyzed by a tool implemented in Java that performs an automatic search for performance problems based on JavaPSL, a Java version of ASL.

6 Summary and Future Work

In this paper we demonstrated the viability of automated performance analysis based on performance property specifications. We described the general idea and explained the structure of the the Apart Specification Language (ASL). In ASL, performance properties are specified with respect to a specific data model.

We presented our data model for shared-memory parallel code that allows the representation of performance critical data such as time waited to enter a construct. The data model is designed according to restrictions on what can actually be measured from the execution of OpenMP program. We rely on the POMP specification [10] and source-to-source instrumentation added by Opari [11] to expose OpenMP execution events, since OpenMP still lacks a standard profiling interface.

We tested the efficacy of our property specification on some test applications. To acquire the performance data needed for the data model, we relied on our own OpenMP profiler `ompP` [6]. The examples show that the data required for checking the properties can be derived from the execution of an OpenMP program and that the performance properties given in examples in Sect. 3 are able determine the major cause of inefficiency in the presented example programs.

Tuning hints can be associated with performance properties. Currently only the name of the property (e.g., `CriticalSectionContention`) conveys information on the reason of the detected inefficiency. However, it is easy to augment the property specification with a more elaborate explanation of the detected inefficiency and to give advice with respect to tuning options to the application developer.

Future work is planned along several directions. First, the set of performance properties can be extended. For example, data coming from hardware performance counters are not yet included, especially cache miss counts and instruction rates can give rise to interesting and important properties.

Secondly, we are working on a larger and more automatic environment for performance analysis in the Periscope project [14]. The goal is to have an ASL compiler that takes the specification of performance properties and translates them into C++ code. Compiling the code creates loadable modules used by the Periscope system to detect performance inefficiencies at runtime. Periscope is designed for MPI and OpenMP and supports large scale systems by virtue of a distributed analysis system consisting of *agents* distributed over the nodes large SMP-based cluster systems.

References

1. Mark E. Crovella and Thomas J. LeBlanc. Parallel performance prediction using lost cycles analysis. In *Proceedings of the 1994 Conference on Supercomputing (SC 1994)*, pages 600–609. ACM Press, 1994.
2. Antonio J. Dorta, Casiano Rodríguez, Francisco de Sande, and Arturo Gonzáles-Escribano. The OpenMP source code repository. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2005)*, pages 244–250, February 2005.
3. 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.

4. Thomas Fahringer, Michael Gerndt, Graham Riley, and Jesper Larsson Träff. Formalizing OpenMP performance properties with ASL. In *Proceedings of the 2000 International Symposium on High Performance Computing (ISHPC 2000), Workshop on OpenMP: Experience and Implementation (WOMPEI)*, pages 428–439. Springer-Verlag, 2000.
5. Thomas Fahringer and Clóvis Seragiotto Júnior. Automatic search for performance problems in parallel and distributed programs by using multi-experiment analysis. In *Proceedings of the 9th International Conference On High Performance Computing (HiPC 2002)*, pages 151–162. Springer-Verlag, 2002.
6. Karl Fürlinger and Michael Gerndt. ompP: A profiling tool for OpenMP. In *Proceedings of the First International Workshop on OpenMP (IWOMP 2005)*, 2005. Accepted for publication.
7. Michael Gerndt. Specification of performance properties of hybrid programs on hitachi SR8000. Technical report, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Institut für Informatik, Technische Universität München, 2002.
8. Clóvis Seragiotto Júnior, Thomas Fahringer, Michael Geissler, Georg Madsen, and Hans Moritsch. On using aksum for semi-automatically searching of performance problems in parallel and distributed programs. In *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003)*, pages 385–392. IEEE Computer Society, February 2003.
9. 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.
10. 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.
11. 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.
12. Bernd Mohr and Jesper Larsson Träff. Initial design of a test suite for automatic performance analysis tools. In *Proc. HIPS*, pages 77–86, 2003.
13. Anna Morajko, Oleg Morajko, Josep Jorba, and Tomàs Margalef. Automatic performance analysis and dynamic tuning of distributed applications. *Parallel Processing Letters*, 13(2):169–187, 2003.
14. Periscope project homepage `http://wwwbode.cs.tum.edu/~gerndt/home/Research/PERISCOPE/Periscope.htm%`.
15. 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.
16. Michael Süß and Claudia Leopold. A user's experience with parallel sorting and openmp. In *Proceedings of the Sixth Workshop on OpenMP (EWOMP'04)*, October 2004.
17. 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.