Evaluating OpenMP Performance Analysis Tools with the APART Test Suite*

Michael Gerndt[1], Bernd Mohr[2], and Jesper Larsson Träff[3]

[1] Institut für Informatik, Technische Universität München, Germany
gerndt@in.tum.de
[2] Forschungszentrum Jülich GmbH, ZAM, Jülich, Germany
b.mohr@fz-juelich.de
[3] C&C Research Labs, NEC Europe Ltd., St. Augustin, Germany
traff@ccrl-nece.de

Abstract. We outline the design of ATS, the *APART Test Suite*, for evaluating (automatic) performance analysis tools with respect to their *correctness* and *effectiveness* in detecting actual performance problems, with focus on the ATS test programs related to OpenMP. We report on results from applying two OpenMP performance analysis tools to the test cases generated from ATS.

1 Introduction

Achieving high performance on parallel computers most often requires performance tuning. The programmer identifies performance problems with the help of manual or automatic performance analysis tools, and transforms the code to improve its performance. The members of the European IST APART working group are developing automatic performance analysis tools for parallel and grid environments. The APART group defined the APART Specification Language (ASL) for writing portable specifications of typical performance problems [2]. Recently, an extensive set of performance properties for hybrid parallel programs combining MPI and OpenMP has been collected [4].

Automatic performance analysis tools, such as those implemented by APART members [1, 5, 7, 8], must be tested with respect to *correctness* and *effectiveness* in detection of actual performance problems. For an automatic performance analysis tool *(positive) correctness* means that the tool is able to detect manifested performance problems in a given application; negative correctness means that the tool does not falsely report performance problems where none exist. To aid the correctness testing and to provide a "standardized" testbed that can be applied to different tools, we are developing the *APART Test Suite (ATS)* framework which allows for easy construction of synthetic positive and negative

* Part of this work is funded by the European Commission via the working group on Automatic Performance Analysis: Real Tools (APART),
http://www.fz-juelich.de/apart/

Table 1. OpenMP performance properties related to synchronization.

`critical_section_locking`	critical section overhead without competing threads
`critical_section_contention`	critical section overhead with competing threads
`serialization_due_to_critical_section`	all work in parallel loop in critical section
`frequent_atomic`	excessive time spent in simple atomic operation
`setting_lock`	overhead for setting a lock without competition
`lock_testing`	overhead for lock testing
`lock_waiting`	overhead for waiting for a lock
`all_threads_lock_contention`	locking overhead due to contention
`pairwise_lock_contention`	locking overhead by pairs of threads

test programs. The current version includes test cases for MPI and OpenMP performance properties in C and Fortran [6].

This paper gives an overview of typical OpenMP performance properties (Section 2), explains the basic structure of ATS (Section 3) and reports on first findings of an evaluation study of two OpenMP performance tools (Section 4).

2 A Performance Property Hierarchy for OpenMP

To describe performance properties of parallel programs an object-oriented, functional formalism called ASL (APART Specification Language) has been developed [2]. In ASL terminology a *performance property* characterizes a particular performance-related behavior of a program based on available or required *performance data*.

Performance data is primarily *dynamic information* collected during one or more sample runs of the program, and can be either trace or summary information. Performance data, however, also includes *static information* about the program (block structure, program and data flow information, loop scheduling

Table 2. OpenMP load imbalance performance properties.

`imbalance_in_parallel_region`	different amount of work per thread
`imbalance_at_barrier`	different arrival time at explicit barrier
`imbalance_in_parallel_loop`	different amount of work in iterations
`imbalance_in_parallel_loop_nowait`	imbalanced parallel loop without implicit barrier
`imbalance_in_parallel_section`	different amount of work in parallel sections
`imbalance_due_to_uneven_section_distribution`	more sections than threads, some threads executed multiple sections
`imbalance_due_to_not_enough_sections`	less sections than threads
`unparallelized_in_master_region`	idle threads due to OpenMP master region
`unparallelized_in_single_region`	idle threads due to OpenMP single region
`unparallelized_in_ordered_loop`	thread serialization
`imbalance_in_ordered_loop`	different amounts of work in ordered region

Table 3. OpenMP performance properties related to control of parallelism.

`dynamic_scheduling_overhead`: scheduling overhead due to dynamic scheduling
`scheduling_overhead_in_parallelized_inner_loop`: inner loop with few iterations was parallelized even though outer loop has much more iterations
`insufficient_work_in_parallel_loop`: loop overhead dominates execution
`firstprivate_initialization`: overhead for initialization of firstprivate variables
`lastprivate_overhead`: initialization overhead of lastprivate variables
`reduction_handling`: overhead for reduction operation handling

Table 4. OpenMP performance properties related to inefficient serial execution.

`false_sharing_in_parallel_region`: overhead for access to different array elements on same cache line

information etc.) and the programming model/paradigm. Examples of performance properties are load imbalance, abundant or mis-scheduled communication, and cache misses.

A performance property is described by a boolean *condition*, and has an associated *severity* for expressing the relative importance of the property. A performance property is a *performance problem* if it is present and its severity exceeds a preset threshold. A *performance bottleneck* is a most severe performance problem. In this framework performance engineering consists in locating and eliminating performance bottlenecks.

Using the ASL formalism, the APART group has compiled hierarchically structured specifications of typical performance properties for the programming paradigms MPI, OpenMP, and HPF [2]. Specifications for OpenMP can be found in [3], and considerably more detailed in [4]. We briefly summarize the properties recorded in [4], since these are the properties that are closely mirrored in ATS. The specification is divided into four categories: (i) *synchronization*, (ii) *load imbalance*, (iii) *control of parallelism*, and (iv) *inefficient serial execution*. Concrete properties in the four categories are listed in Tables 1 to 4. For hybrid OpenMP/MPI programming, additional categories contain properties related to MPI communication and to parallel I/O.

3 APART Test Suite Design

We briefly describe the design of the APART Test Suite, especially as pertaining to the OpenMP properties listed in the previous section. The first version of ATS covers the "standard" parallel programming paradigms MPI and OpenMP, but the modular structure of the design easily allows to add modules for other programming paradigms like HPF, PVM, or POSIX threads.

The main idea of our design is a collection of functions with a standardized interface, hierarchically organized into modules that can be easily combined to produce a program exhibiting desired performance properties. Thus, functions

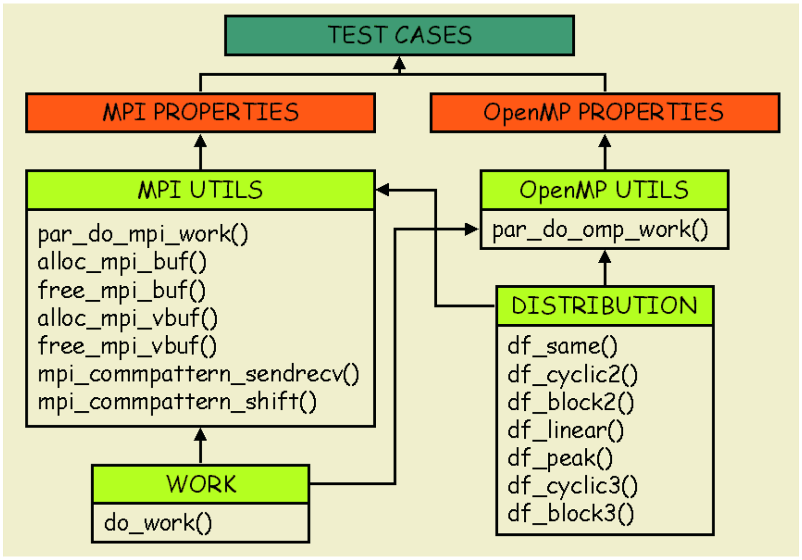


Fig. 1. Basic Structure of the ATS framework.

from the modules should have as little context as possible, and whatever context is necessary is provided through standardized parameters. Furthermore, since (automatic) performance analysis tools have different thresholds/sensitivities, it is important that the test suite is parametrized so that the relative severity of the properties can be controlled by the user.

Figure 1 shows the basic structure of the ATS framework for the MPI and OpenMP programming paradigms. The boxes with shaded titles represent the basic modules. Arrows indicate *used-by* relationships. For the lower levels, the functions provided by those module are listed.

The lowest two modules, `work` and `distribution`, provide basic functionality to specify the amount of generic work to be executed by the individual threads or processes of a parallel program. The next level provides generic support for the two main parallel programming paradigms MPI and OpenMP. The third level implements *property functions* which when executed exhibit one specific performance property. For OpenMP this means that we implemented one such function for each property listed in Section 2. Finally, there are several ways of calling the property functions so that different aspects of performance correctness testing can be addressed. For a full description of the ATS framework, see [6].

4 Evaluation of OpenMP Performance Tools

We used the ATS to evaluate four OpenMP performance analysis tools, namely the Hitachi Profiling Tool `pmfunc` specifically for Hitachi parallel supercomputers, Intel's performance tool Vtune, the platform-independent tools EXPERT and Vampir. Vtune, Vampir and the Hitachi Profiling Tool are manual, while EXPERT is an automatic tool. Due to limited space, we only discuss `pmfunc` and EXPERT below.

4.1 Hitachi Profiling Tool

The Hitachi SR8000 supercomputer is a clustered SMP design. Each node consists of eight processors that can be used by applications. Applications are developed in the hybrid programming model, MPI across nodes and OpenMP or COMPASS, a Hitachi proprietary shared memory programming API, within nodes.

The Fortran, C and C++ compilers on the Hitachi SR8000 can automatically instrument program regions. A compiler switch `pmfunc` directs the compiler to instrument user functions. The switch `pmpar` instruments all COMPAS parallel regions, independent of whether they are generated by automatic parallelization or by manual transformation. In OpenMP programs, the switch instruments only `OMP PARALLEL REGION`, `OMP PARALLEL DO`, and `OMP PARALLEL SECTION`. It does not instrument work-sharing constructs within parallel regions. The compiler switches not only insert calls to the monitoring routines, but also link a performance monitoring library to the instrumented code. This library measures for each instrumented region, that is, user function or parallel region, among other values execution time, cache misses, load/store instructions, floating point operations, and number of executions. See [9] for details.

For each node a separate information file is generated that can be inspected with the `pmpr` command. This command displays the information contained in the output files in a humanly readable form (Figure 2).

The information measured for a function or parallel region is presented separately for each thread. This makes it possible to investigate differences among

```
imbalance_due_to_uneven_section_distribution[2](omp_pattern.c+560)
    CPU time  FLOP     Inst    LD/ST  D-cache   MFLOPS      MIPS   Times
------------------------------------------------------------------------
IP0    4.492<   16>  76903k>  36190k>   2272k    0.000>   17.120>     4>
IP1    4.492    16>  76903k>  36190k>   2273k>   0.000    17.120      4>
IP2    4.493>   16>  76903k>  36190k>   2272k    0.000    17.116      4>
IP3    4.492    16>  76903k>  36190k>   2272k    0.000    17.119      4>
IP4    4.493     8<  38452k<  18095k<   1136k<   0.000<    8.559<     4>
IP5    4.492     8<  38452k<  18095k<   1137k    0.000     8.559      4>
IP6    4.493     8<  38452k<  18095k<   1137k    0.000     8.559      4>
IP7    4.493     8<  38452k<  18095k<   1137k    0.000     8.559      4>
------------------------------------------------------------------------
TOTAL 35.940    96  461419k  217138k   13635k    0.000   102.693     32
------------------------------------------------------------------------
Element parallelizing rate : (TOTAL)/(Max * IPs)
     CPU time :  99.98[%] =   35.940/(4.493168*8)
     FLOP     :  75.00[%] = 96/(16*8)
```

Fig. 2. Example information file showing load imbalance due to uneven section distribution. The code had 12 sections, so the first four threads got two sections. The evaluation program `pmpr` marks the largest and smallest values in each column with '<' and '>' respectively.

the threads, for example, resulting from load imbalance. For parallel regions the values can be compared directly. For functions, the data have to be interpreted more carefully. If a function is started on the master processor but includes parallel regions, the data of the other threads are accumulated in the master processor. If, on the other hand, a function is called in a parallel region, the execution information is reported for each thread individually.

Since execution time and instruction counts are given on a per thread basis for parallel regions, load imbalance properties could be identified. The execution time did show the imbalance only in `imbalance_in_parallel_loop_nowait` since in all other cases the implicit barrier ensures equal execution times. In those cases, the imbalance was detected from the differences in the instruction counts.

The difference between `unparallelized_ordered_loop` and `imbalance_in_-ordered_loop` was not shown since ordered loops are executed by the Hitachi compilers as sequential loops.

The only test case based on cache misses, `false_sharing_in_parallel_-region`, could be detected from a very high cache miss rate (about 75%). The tool did not give any indication that the misses resulted from false sharing.

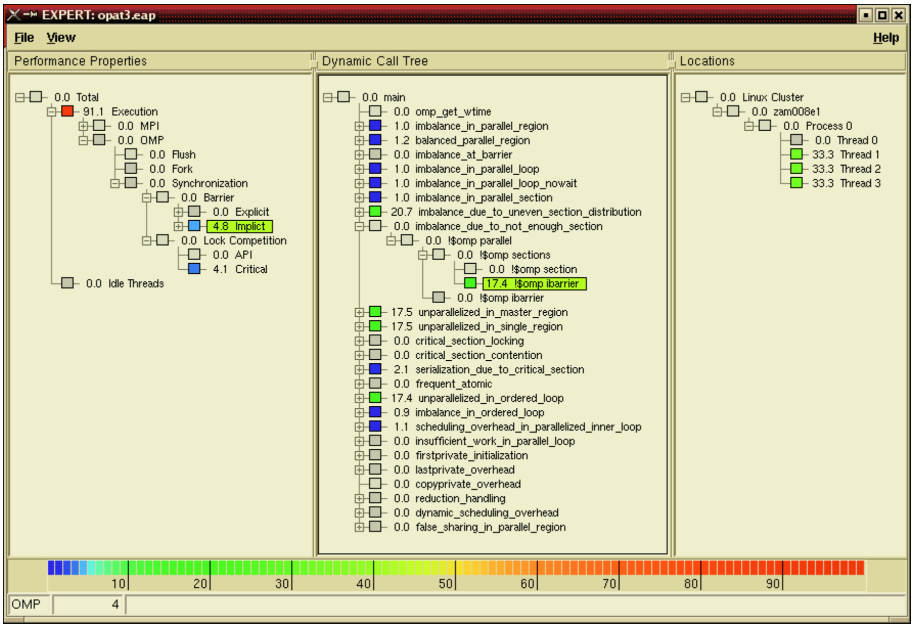
Properties related to synchronization bottlenecks could not be identified since the tool gives no information about synchronization operations. The same is true for properties checking parallelism overhead.

4.2 EXPERT

The EXPERT automatic event trace analyzer [8] is part of the KOJAK project (Kit for Objective Judgment and Knowledge-based Detection of Performance Bottlenecks), whose aim is a generic automatic performance analysis environment for parallel programs. Performance problems are specified in terms of *execution patterns* that represent situations of inefficient behavior. These are input to an analysis process that recognizes and quantifies inefficient behavior in event traces. The pattern specification in EXPERT is different from ASL, as it allows to specify how performance metrics are calculated out of basic event attributes.

The KOJAK analysis process is composed of two parts: a semi-automatic multi-level instrumentation of the user application followed by an automatic analysis of the generated performance data. Running an instrumented executable generates a trace file in the EPILOG format. After program termination, the trace file is fed into the EXPERT (Extensible Performance Tool) analyzer. The analyzer generates an analysis report, which serves as input for the EXPERT presenter. A screen dump is shown in Figure 3. Using the color scale shown on the bottom, the severity of performance problems found (left pane) and their distribution over the program's call tree (middle pane) and machine locations (right pane) is displayed. By expanding or collapsing nodes in each of the three trees, the analysis can be performed on different levels of granularity.

The experiments were performed on a 4 CPU Intel IA32 Linux system. As can be seen in the left part of Figure 3, for OpenMP, EXPERT is currently able to identify performance problems related to Flush, thread startup (Fork), barrier and locking overhead. It also shows that EXPERT could detect all properties



```
0.0 Total
  91.1 Execution
    0.0 MPI
    0.0 OMP
      0.0 Flush
      0.0 Fork
      0.0 Synchronization
        0.0 Barrier
          0.0 Explicit
          4.8 Implicit
        0.0 Lock Competition
          0.0 API
          4.1 Critical
  0.0 Idle Threads
```

```
0.0 main
  0.0 omp_get_wtime
  1.0 imbalance_in_parallel_region
  1.2 balanced_parallel_region
  0.0 imbalance_at_barrier
  1.0 imbalance_in_parallel_loop
  1.0 imbalance_in_parallel_loop_nowait
  1.0 imbalance_in_parallel_section
  20.7 imbalance_due_to_uneven_section_distribution
  0.0 imbalance_due_to_not_enough_section
    0.0 !$omp parallel
      0.0 !$omp sections
        0.0 !$omp section
        17.4 !$omp ibarrier
      0.0 !$omp ibarrier
  17.5 unparallelized_in_master_region
  17.5 unparallelized_in_single_region
  0.0 critical_section_locking
  0.0 critical_section_contention
  2.1 serialization_due_to_critical_section
  0.0 frequent_atomic
  17.4 unparallelized_in_ordered_loop
  0.9 imbalance_in_ordered_loop
  1.1 scheduling_overhead_in_parallelized_inner_loop
  0.0 insufficient_work_in_parallel_loop
  0.0 firstprivate_initialization
  0.0 lastprivate_overhead
  0.0 copyprivate_overhead
  0.0 reduction_handling
  0.0 dynamic_scheduling_overhead
  0.0 false_sharing_in_parallel_region
```

```
0.0 Linux Cluster
  0.0 zam008e1
    0.0 Process 0
      0.0 Thread 0
      33.3 Thread 1
      33.3 Thread 2
      33.3 Thread 3
```

Fig. 3. Expert Presenter result display. Left pane shows performance problems, middle pane their distribution, and right pane their machine location.

related to load imbalance, as indicated by darker boxes in the middle pane. The numbers shown are the percentage of the total execution time lost because of the identified performance property. The load imbalance problems are detected due to high barrier overhead. Similar properties could be defined to distinguish lock contention and frequent locking. By selecting specific call tree nodes the distribution of the problem over the machine, processes, and threads can easily be investigated in the right pane (shown for the property function `imbalance_due_to_uneven_section_distribution`). EXPERT was also able to identify performance problems related to locks and critical regions (not visible in Figure 3).

5 Conclusion and Future Work

We listed the current set of OpenMP performance property functions in the APART Test Suite (ATS), a framework which can be used to generate test cases for the evaluation of (automatic) performance analysis tools. We ran the full set of OpenMP functions with a semi-automatic vendor performance analysis tool and the automatic EXPERT tool.

The effectiveness of the tools depends highly on the information provided by the runtime monitor. With the Hitachi tool no synchronization information is available. Thus, even simple load imbalance problems cannot be easily detected. The hardware counter information can be used instead to get hints to load imbalances and have proved very useful to identify false sharing. The Hitachi profiling tool provides only summary tables in ASCII form. The EXPERT tool detects performance problems automatically, for example code regions with high synchronization overhead, but more detailed properties explaining the reason for load imbalance cannot be detected automatically.

The ATS enabled us to evaluate the four tools (results for Vtune and Vampir are not discussed here due to limited space). Due to the well defined semantics of the property functions, the strength and weaknesses of the different tools can be easily identified. However, a formal comparison or ranking of the tools is quite difficult since, except for EXPERT, the user has to interpret the information provided by the tools and identify the performance properties manually.

We plan to extend ATS in the future with more performance properties and to work on the automatic generation of test programs combining the individual modules. ATS is freely available for the evaluation of other performance tools at `http://www.fz-juelich.de/apart/ats/`.

References

1. A. Espinosa. *Automatic Performance Analysis of Parallel Programs.* PhD thesis, Universitat Autonoma de Barcelona, 2000.
2. T. Fahringer, M. Gerndt, B. Mohr, F. Wolf, G. Riley, and J. L. Träff. Knowledge specification for automatic performance analysis. Technical Report FZJ-ZAM-IB-2001-08, Forschungszentrum Jülich, 2001.
3. T. Fahringer, M. Gerndt, G. Riley, and J. L. Träff. Formalizing OpenMP performance properties with ASL. In *Workshop on OpenMP: Experience and Implementations (WOMPEI), Intl. Symposium on High Performance Computing (ISHPC2K)*, LNCS 1940, pp. 428–439, 2000.
4. M. Gerndt. *Specification of Performance Properties of Hybrid Programs on Hitachi SR8000.* Peridot Technical Report, TU München, 2002
5. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvine, K. L. Karavanic, K. Kunchithapadam, T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.
6. B. Mohr and J. L. Träff. Initial Design of a Test Suite for (Automatic) Performance Analysis Tools. In *8th Intl. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2003)*, pp. 77–86, 2003.
7. H.-L. Truong, T. Fahringer. SCALEA: A Performance Analysis Tool for Distributed and Parallel Programs. In *Euro-Par 2002*, LNCS 2400, pp. 75–85, 2002.
8. F. Wolf, B. Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. *Journal of Systems Architecture*, 49(10-11):421–439, 2003. Special Issue "Evolutions in parallel distributed and network-based processing".
9. `www.lrz-muenchen.de/services/compute/hlrb/manuals`