

Evaluating OpenMP Performance Analysis Tools with the APART Test Suite*

Michael Gerndt
Institut für Informatik, LRR
Technische Universität München
gerndt@in.tum.de

Bernd Mohr
Forschungszentrum Jülich GmbH
ZAM, 52425 Jülich, Germany
b.mohr@fz-juelich.de

Jesper Larsson Träff
C&C Research Labs, NEC Europe Ltd.
Rathausallee 10, 53757 St. Augustin, Germany
traff@ccrl-nece.de

Abstract

The APART working group is developing the APART Test Suite (ATS) for evaluating (automatic) performance analysis tools with respect to their *correctness* – that is, their ability to detect actual performance problems – and their effectiveness in doing so. This article briefly describes the design of the test suite and in more detail the ATS test programs related to OpenMP. It reports on results from applying two manual and one automatic OpenMP performance analysis tools to the test cases generated from ATS.

1 Introduction

Achieving high performance on parallel computers most often requires manual performance tuning. The programmer identifies performance problems with the help of interactive or automatic performance analysis tools, and transforms the code to improve its performance.

It is the goal of the members of the European IST APART working group to develop 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 [3]. The most recent specification [5] was done in the Peridot project¹ for the Hitachi SR8000 supercomputer at the Leibniz Computer Centre in Munich. It provides an extensive set of performance properties for hybrid parallel programs combining MPI and OpenMP.

Automatic performance analysis tools, as those currently under development by the groups in APART [2, 7, 12, 13, 14, 15], of course need to be tested with respect to their *correctness* and *effectiveness*. 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 this end, we are developing the *APART Test Suite (ATS)* framework which allows the easy construction of synthetic positive and negative test programs. The current version includes test cases for MPI and OpenMP performance properties in C and Fortran.

This article gives an overview of the typical OpenMP performance properties (Section 2). It explains the basic structure of the ATS framework and presents examples of typical test programs (Section 3). Section 4 reports on first findings of an evaluation study of state-of-the-art OpenMP performance analysis tools currently being performed by the authors. We report on results from the application of two manual and one automatic OpenMP performance analysis tools to the test cases generated from ATS.

*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>

¹www.in.tum.de/peridot

2 A Performance Property Hierarchy for OpenMP

To describe performance properties of parallel programs a formalism called ASL (APART Specification Language) was previously developed by the APART group [3]. ASL uses the following 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 information etc.) and the programming model/paradigm. Examples of such performance properties are load imbalance, abundant or mis-scheduled communication, 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 the most severe performance problem. In this framework performance engineering consists in locating and eliminating performance bottlenecks.

ASL is an object-oriented, functional formalism for describing performance data and properties. Properties are described separately (i.e. interdependence or subsumption of properties are not expressed explicitly in the ASL formalism), but specifications can be made very compact by the use of *property templates*, which allows to define *meta properties*. Metaproperties specify new properties by combining already defined properties.

In addition to the ASL formalism, the APART group has compiled hierarchically structured specifications of typical performance properties for the MPI, OpenMP, and HPF programming paradigms [3]. Specifications for OpenMP can be found in [4], and more elaborated in [5]. We briefly summarize the properties recorded in [5], since these are the properties that are closely mirrored in ATS. The specification is divided into four categories:

1. Synchronization
2. Load imbalance
3. Control of parallelism
4. Inefficient serial execution

For hybrid OpenMP/MPI programming, additional categories contain properties related to MPI communication and to I/O. The concrete properties currently in the four categories are listed in Tables 1 to 4.

3 APART Test Suite Design

We now 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 and hierarchically organized into modules that can be easily combined to produce a program exhibiting desired performance properties. Thus, functions from the modules should have as little context as possible, and whatever context is necessary is provided through the 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

Table 1: OpenMP performance properties related to synchronization.

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

Table 2: Load imbalance performance properties.

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

Table 3: Performance properties related to control of parallelism.

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

Table 4: Inefficient serial execution.

<code>false_sharing_in_parallel_region</code>	: overhead for access to different array elements on same cache line
---	--

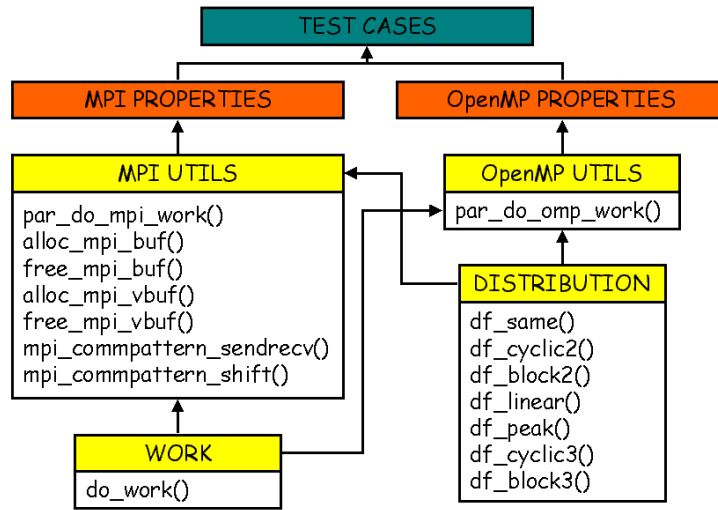


Figure 1: Basic Structure of the ATS framework.

testing can be addressed. In the following, we describe the modules in more detail. For a full description of the ATS framework, see [9].

3.1 Specification of Work

Abstractly, an application consists of two parts: the actual desired user computation (*work*) and the necessary extra communication, synchronization, and organization overhead for executing the program in parallel. Therefore, for a generic test program we first need a way to specify a generic piece of work.

The current ATS version provides a `do_work` function which allows to specify “work” in terms of desired execution time. The real time spent in `do_work` is only approximated up to a certain degree (approx. milliseconds) and is not guaranteed to be stable, especially under heavy system load. Other work functions could attempt to simulate specific characteristics of the computational work, e.g. memory access patterns, cache misses, and so on.

Based on the `do_work` function, parallel versions can be implemented for MPI and OpenMP. The parallel work function for OpenMP is shown below:

```
void par_do_omp_work(distr_func_t df, distr_t* dd, double sf) {
    int me, sz;
    me = omp_get_thread_num();
    sz = omp_get_num_threads();
    do_work(df(me, sz, sf, dd));
}
```

The generic specification of parallel work is based on two ideas:

1. The participants are specified implicitly by the active OpenMP thread team, or explicitly through an MPI communicator.
2. The distribution of work among the participants is described through generic distribution parameters: distribution function `df`, distribution descriptor `dd`, and a scale factor `sf`.

The generic work functions are supposed to be called by all participants of a parallel construct, much like a collective communication call in MPI. First, the number of participants in the parallel construct and the rank/ID of the participant within this group is determined. Then, the amount of work for the calling

participant is calculated using the distribution parameters. Finally, the sequential version of `do_work()` is called by each process/thread with the calculated amount of work.

Types of work-distribution (even amount of work, all work done by one process, half-half, etc.) are encoded in distribution functions which take rank/ID, number of processes/threads, scale factor, and distribution descriptor as parameters. Descriptors are specific to the different distribution functions, and are used to hold the specific parameters for the distribution.

The following example shows the `CYCLIC2` distribution where each even process has a predefined low value and each odd process a predefined high value.

```
typedef struct { double low, high; } val2_distr_t;

double df_cyclic2(int me, int sz, double scale, distr_t* dd) {
    val2_distr_t* d = (val2_distr_t*) dd;
    if ( (me % 2) == 0 )
        return d->low * scale;
    else
        return d->high * scale;
}
```

When this distribution is used, for example in `par_do_omp_work`, the distribution descriptor must have been initialized with low and high values. The distribution function `df_cyclic2` returns the low/high value depending on whether the calling process/thread is even or odd, scaled with the scale factor.

ATS provides a number of distribution functions for patterns similar to `CYCLIC2`. If needed, users can provide their own distribution functions and distribution descriptors, as long as their implementation follows the same basic principles and the signature of the distribution function is equivalent to `distr_func_t`.

3.2 Performance Property Functions

Using the functions for work and distribution, it is now quite simple to implement functions which, when executed, exhibit a well-defined performance behavior. Ideally, for each of the performance properties listed in [3], at least one function should be defined. Therefore, we call them *ATS performance property functions*.

Of course, these functions should be implemented as generic as possible, i.e., there should be no restrictions on the context where the functions are called (e.g., the number of threads). Also, there should be parameters to describe the exact form and the severity of the performance property.

A large portion of the performance property functions is related to an imbalance of work or communication. In this case, the property functions typically have three parameters: Through a distribution function and its corresponding distribution descriptor the imbalance of work or communication is specified. A third parameter describes how often the main body of the performance property function should be repeated. As an example, the complete source code of the performance property function `imbalance_at_barrier()` is shown below:

```
void imbalance_at_barrier(distr_func_t df, distr_t* dd, int r) {
    int i;
    #pragma omp parallel private(i)
    {
        for (i=0; i<r; ++i) {
            par_do_omp_work(df, dd, 1.0);
            #pragma omp barrier
        }
    }
}
```

The amount and type of imbalance is controlled by the distribution function `df` and its parameters `dd`.

3.3 Single Performance Property Testing

Based on the collection of performance property functions, test programs can now be built for individual testing of each of the performance properties. Such test programs are generated automatically from the performance property function signatures, by using the parser toolkit PDT [6]. Based on the information about the function argument types, it is easy to generate a main program skeleton which reads the necessary property parameters from the command line and then calls the property function after initializing parallel execution if necessary. More extensive experiments based on these synthetic test programs can then be executed through scripting languages or automatic experiment management systems, such as ZENTURIO [11].

4 Evaluation of OpenMP performance tools

We used the ATS to evaluate three OpenMP performance analysis tools, namely the Hitachi Profiling Tool `pmfunc` specifically for Hitachi parallel supercomputer, and the platform-independent tools `EXPERT` and `Vampir`. `Vampir` and `pmfunc` are manual performance tools, while `EXPERT` is an automatic tool.

4.1 Hitachi Profiling Tool

The Hitachi SR8000 supercomputer is a clustered SMP design. Each node consists of nine processors, eight of which can be used by applications. The Hitachi SR8000 at LRZ München has 168 nodes and a peak performance of more than 2 TFlops. Applications are developed in the hybrid programming model, MPI across nodes and OpenMP or COMPASS, the Hitachi proprietary shared memory programming API, within a node.

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 `pmpr` 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 do not only insert calls to the monitoring routines, but do also link a standard performance monitoring library to the instrumented code. This standard library measures for each instrumented region, i.e., user function or parallel region, among other values execution time, cache misses, load/store instructions, floating point operations, and number of executions. For details see www.lrz-muenchen.de/services/compute/hlrb/manuals.

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 human readable form (Figure 2).

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

Not all of the test cases could be executed on the Hitachi. The `copyprivate_overhead` kernel demonstrating high overhead for copying back data in a single region to all private copies of the other threads could not be compiled.

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

```

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)

```

Figure 2: These data show the load imbalance due to an uneven section distribution. The code had 12 sections. Thus the first four threads got two sections. The evaluation program `pmpr` marks the largest and smallest values in each column with '`<`' and '`>`' respectively.

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 prototype automatic event trace analyzer [14, 15] is part of the KOJAK project (Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks). KOJAK aims at developing 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 patterns are input for an analysis process that recognizes and quantifies the inefficient behavior in event traces. EXPERT's pattern specification is different from the ASL specification, 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. To begin the process, the user supplies the application's source code, written in either C, C++, or Fortran, to OPARI (OpenMP Pragma And Region Instrumentor), which performs automatic instrumentation of OpenMP constructs and redirection of OpenMP-library calls to instrumented wrapper functions on the source-code level based on the POMP API [8]. Instrumentation of user functions is done either on the source-code level using TAU or using a compiler-supplied profiling interface. Instrumentation for MPI events is accomplished using a PMPI wrapper library, which generates MPI-specific events by intercepting calls to MPI functions. All MPI, OpenMP, and user-function instrumentation call the EPILOG (Event Processing, Investigating and LOGging) run-time library, which provides mechanisms for buffering and trace-file creation. At the end of the instrumentation process the user has a fully instrumented executable.

Running this 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. Figure 3 shows a screendump of the EXPERT presenter. 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.

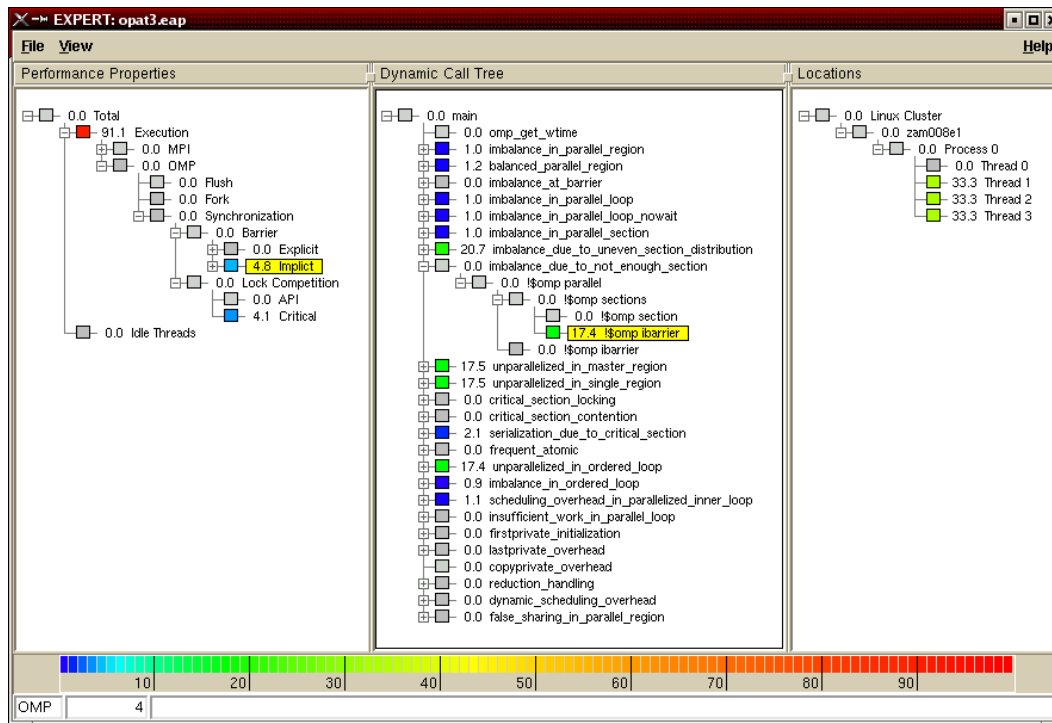


Figure 3: Expert Presenter result display. The left pane shows performance problems, middle pane their distribution, and right pane their machine location.

KOJAK runs on a variety of platforms (Cray, IBM, SGI, Sun, Hitachi, Linux); the experiments were performed on a 4 CPU Intel IA32 system running SuSE Linux. OpenMP compilation was done using the PGI compilers version 4.0 which do not yet fully support OpenMP 2.0, so the `copyprivate overhead` kernel could not be compiled.

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.

The screendump shows, that EXPERT detected all properties related to load imbalance (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. EXPERT could be extended to distinguish the different reasons for imbalance. 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 the example screendump).

The current prototype of EXPERT does not yet support definition of performance properties based on hardware counter information. Thus it is not possible to detect problems like false sharing.

4.3 Vampir

Vampir [1] is the industry standard tool for the analysis of parallel MPI applications based on sophisticated visualizations of measured execution traces. However, the visualization provided by Vampir is independent from the semantics of traces, so it can also be used to visualize and analyze other sorts of measurements. The KOJAK environment (see Section 4.2) also provides a tool which converts KOJAK's EPILOG trace format into a format suitable for Vampir. The conversion maps OpenMP constructs into Vampir symbols and activities

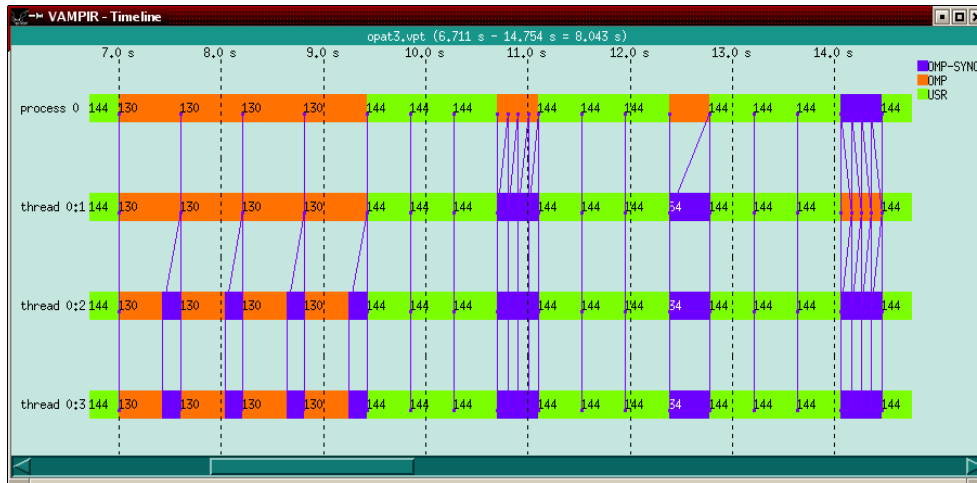


Figure 4: Vampir time line display.

and OpenMP barriers into a Vampir collective operation. This allows not only to see the dynamic behavior of an OpenMP application using a Vampir timeline diagram but also to use Vampir's powerful filter and selection capabilities to generate various execution statistics for the phases of the OpenMP application. In addition, all source code information contained in a trace is preserved during conversion; this allows to display the corresponding source code by simply clicking on the desired activity.

In principle, as the execution traces are generated by the KOJAK measurement system, it is possible to identify the same performance properties as for EXPERT. The main difference is that users have to locate and identify the performance problems for themselves by browsing through the diagrams provided by Vampir. Problems related to synchronization and thread startup can easily be diagnosed by investigating the execution statistics for the corresponding activities. Severe instances of the problems can also easily be located in the time line display.

Figure 4 shows the execution of OpenMP performance property test functions monitored using the KOJAK measurement infrastructure and visualized by Vampir; namely (from left to right) `imbalance_due_to_uneven_section_distribution`, `imbalance_due_to_not_enough_section`, `unparallelized_in_master_region`, and `unparallelized_in_single_region`.

The load imbalance introduced by the performance properties can easily be located by looking at the distribution of the activity `OMP-SYNC` (shown in dark color in the example screendump). The same is true for waiting times introduced by locks or critical regions.

5 Conclusion and Future Work

We outlined the design of the APART Test Suite (ATS) and its current set of OpenMP performance property functions. We ran the full OpenMP suite with two manual and one automatic tool. Other tools will be evaluated in the future such as Intel's Thread Profiler.

The effectiveness of the tools depends highly on the information provided by the runtime monitor. On the Hitachi, in contrast to both other tools, no synchronization information is available. Thus, even simple load imbalance problems, such as a master region, cannot be easily detected. The hardware counter information can be used instead to get hints to load imbalances and proved very useful to identify false sharing.

The Hitachi profiling tool provides only summary tables in ASCII form. Vampir visualizes the dynamic behavior and leaves the identification of performance problems to the user. EXPERT, in contrast, detects performance problems automatically. For example, it detects code regions with high synchronization overhead. None of the tools is able to identify more detailed properties explaining the reason for load imbalance automatically.

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. We would like to invite everybody to evaluate its own performance tools with the test suite available at <http://www.fz-juelich.de/apart/ats/> in C and Fortran. Feedback, especially suggestions for new performance properties, are highly welcome and will be included in the test suite.

References

- [1] H. Brunst and M. Winkler and W. E. Nagel and H.-Ch. Hoppe. Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach. In *Intl. Conference on Computational Science (ICCS 2001)*, Part II, LNCS 2074, pp. 751–760, 2001.
- [2] A. Espinosa. *Automatic Performance Analysis of Parallel Programs*. PhD thesis, Universitat Autònoma de Barcelona, 2000.
- [3] 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.
- [4] 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.
- [5] M. Gerndt. *Specification of Performance Properties of Hybrid Programs on Hitachi SR8000*. Peridot Technical Report, TU München, 2002
- [6] K. A. Lindlan, J. Cuny, A. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. In *Supercomputing 2000*. <http://www.sc2000.org/proceedings/techpaper/papers/pap.pap167.pdf>
- [7] 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.
- [8] B. Mohr, A. Malony, S. Shende, and F. Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. *The Journal of Supercomputing*, 23:105–128, 2002.
- [9] 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.
- [10] www.lrz-muenchen.de/services/compute/hlrb/manuals
- [11] R. Prodan, T. Fahringer. ZENTURIO: An Experiment Management System for Cluster and Grid Computing. In *4th Intl. Conference on Cluster Computing (CLUSTER 2002)*, pp. 9–18, 2002.
- [12] H.-L. Truong, T. Fahringer, G. Madsen, A. D. Malony, H. Moritsch, S. Shende. Using SCALEA for Performance Analysis of Distributed and Parallel Programs. In *Supercomputing 2001*. <http://www.sc2001.org/papers/pap.pap243.pdf>
- [13] 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.
- [14] F. Wolf. *Automatic Performance Analysis on Parallel Computers with SMP Nodes*. Dissertation, NIC Series, Vol. 17, Forschungszentrum Jülich, 2002.
- [15] F. Wolf, B. Mohr. Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. In *11th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pp. 13–22, 2003.