

An Implementation of the POMP Performance Monitoring Interface for OpenMP Based on Dynamic Probes

Luiz DeRose
IBM Research
ACTC
Yorktown Heights, NY, USA
laderose@us.ibm.com

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

Seetharami Seelam
University of Texas
at El Paso
El Paso, TX, USA
seelam@cs.utep.edu

Abstract. OpenMP has emerged as the standard for shared memory parallel programming. Unfortunately, it does not provide a standardized performance monitoring interface, such that users and tools builders could write portable libraries for performance measurement of OpenMP programs. In this paper we present an implementation of a performance monitoring interface for OpenMP, based on the POMP proposal, which is built on top of DPCL, an infrastructure for binary and dynamic instrumentation. We also present overhead measurements of our implementation and show examples of utilization with two versions of POMP compliant libraries.

1 Introduction

OpenMP has emerged as the standard for shared memory parallel programming, allowing users to write applications that are portable across most shared memory multiprocessors. However, application developers still face a large number of application performance problems, which make it harder to achieve high performance on these systems. Moreover, these problems are difficult to detect without the help of performance tools.

Unlike MPI which includes a standard monitoring interface (PMPI), OpenMP does not provide yet a standardized performance monitoring interface. In order to simplify the design and implementation of portable OpenMP performance tools, Mohr et. al. [10] proposed POMP, a performance monitoring interface for OpenMP. This proposal extends experiences of previous implementations of monitoring interfaces for OpenMP [1, 8, 12], and is under review for standardization by the OpenMP architecture review board.

In this paper we present a POMP implementation based on dynamic probes. This implementation is built on top of DPCL [5], an object-based C++ class library and run-time infrastructure, developed by IBM, which is based on the Paradyn [9] dynamic instrumentation technology from the University of Wisconsin. To the best of our knowledge, this is the first implementation of the current POMP proposal, as well as the first implementation based on binary modification, instead of a compiler or pre-processor based one. It extends previous experiences with binary instrumentation of OpenMP programs from OMPTrace [3] and CATCH [7].

The advantage of this approach lies in its ability to modify the binary with performance instrumentation, without requiring access to the source code or re-compilation, whenever a new set of instrumentation is required. This is in contrast to the most common instrumentation approach, which augments source code statically with calls to specific instrumentation libraries (e.g., TAU [11], Pablo [13], SvPablo [6], and the HPM Toolkit [4]). In addition, since it relies only on the binary, this POMP implementation is programming-language independent. Moreover, it will provide a validation infrastructure for the POMP proposal, which has as one of its main goals the definition of a generic and portable monitoring interface for OpenMP.

The remainder of this article is organized as follows: In Section 2 we briefly describe the main DPCL features, as well as the IBM compiler and run-time library issues that make our dynamic instrumentation tool for POMP possible. In Section 3 we describe our experiences in implementing our tool. In Section 4 we present examples of utilization of our POMP monitoring library implementations.

2 A Dynamic Instrumentation Tool for POMP

2.1 The Dynamic Probe Class Library

Dynamic instrumentation provides the flexibility for tools to insert probes into the binary of applications, without requiring that programs be re-compiled after being instrumented. The Dynamic Probe Class Library, developed at IBM, is an extension of the dynamic instrumentation approach, pioneered by the Paradyn group at the University of Wisconsin [9]. DPCL is built on top of the Dyninst Application Program Interface (API) [2]. Using DPCL, a performance tool can insert code patches into the application binary and start or continue its execution. Access to the source code of the target application is not required and the program being modified does not need to be re-compiled, re-linked, or even re-started.

DPCL provides a set of C++ classes that allows tools to connect, examine, and instrument a spectrum of applications: single processes to large parallel applications. With DPCL, program instrumentation can be done at function entry points, exit points, and call sites. We refer to [5] for a more detailed description of DPCL and its functionality.

2.2 The IBM OpenMP Compiler and Run-time Library

The utilization of DPCL as an infrastructure for a dynamic instrumentation tool for the POMP interface is based on the fact that the IBM compiler translates OpenMP directives into function calls. Figure 1 shows, as an example, the compiler transformations for an OpenMP parallel loop. The OpenMP directive is translated into a call to `xlsmpParDoSetup`, a function from the OpenMP run-time library, which is responsible for thread management, work distribution, and thread synchronization. In addition, an “*outlined*” function containing the body of the OpenMP parallel region is created. This outlined function, (“`A@OL1`” in the example in Figure 1), is called by each of the OpenMP threads.

Since DPCL allows the installation of probes at function call entry and exit points, as well as before and after a function call, POMP probes can be inserted for each parallel region in the target application. As shown in Figure 2 the first pair (DPCL probe (1)) can be inserted before and after the call to the OpenMP runtime library function (corresponding to the POMP parallel enter and exit events), while the second pair (DPCL probe(2)) is inserted at the call entry and exit point of the parallel region (corresponding to the POMP parallel begin and end events). Additionally, a third pair of probes (DPCL probe (3a)) can be inserted at the call entry and exit points or before and after the call site (DPCL probe(3b)) of each function of interest (i.e., representing POMP function begin and end events).

3 Evaluation

In this section, we describe our experiences implementing a tool which allows instrumentation of an OpenMP application with calls to functions of a monitoring library which conforms to the POMP API using binary instrumentation at runtime. The main advantages of this approach are that no special preparations like recompiling or relinking are necessary and that it works independent of the

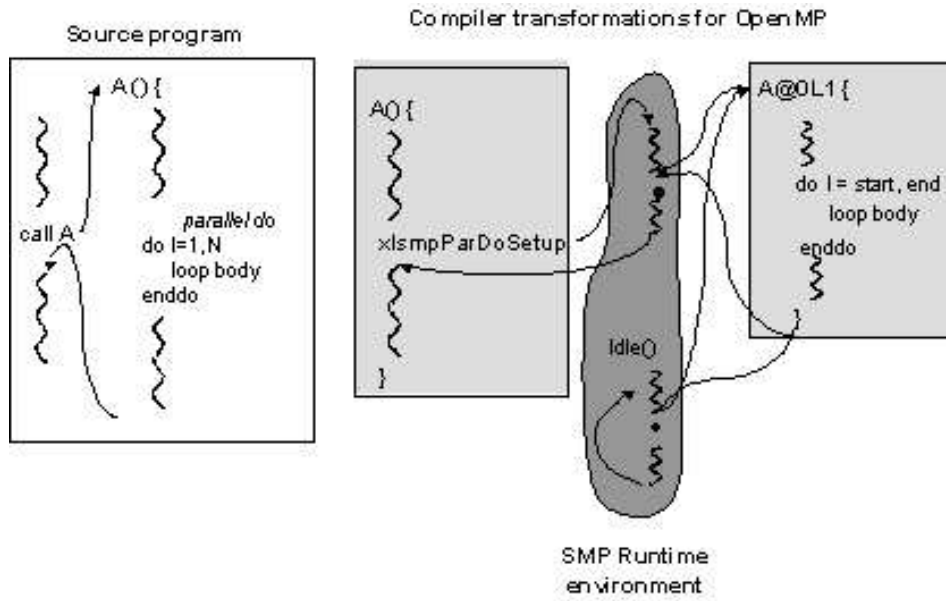


Fig. 1. Compiler transformations for an OpenMP parallel loop.

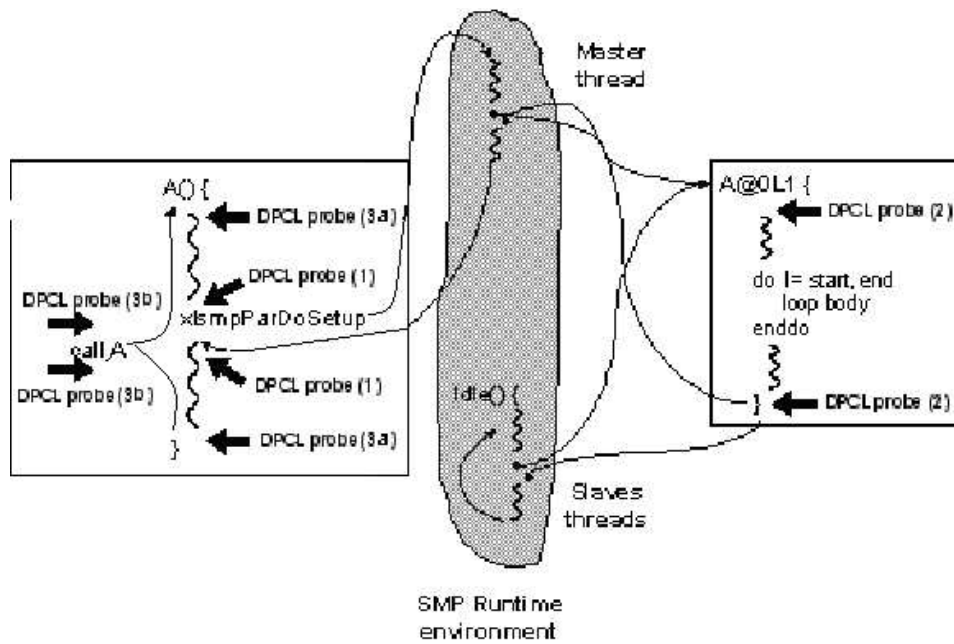


Fig. 2. DPCL probes on functions that contain parallel regions.

programming language used for the OpenMP program. The tool takes a POMP compliant monitoring library in the form of a shared library as well as the OpenMP executable as input parameters. After loading the program, the tool instruments the executable code so that at locations which represents events in the POMP execution model the corresponding POMP monitoring routines are called. From the user's view point, the amount of instrumentation can be controlled through environment variables which describe the level of instrumentation for each group of OpenMP events as proposed by the POMP specification. Instrumentation is also controlled by the set of POMP routines provided by the shared library, i.e., instrumentation is only applied to those events that have a corresponding POMP routine in the shared library. This means that tool builders only need to implement the routines which are necessary for their tool and that instrumentation is minimal (i.e., no calls to unnecessary dummy routines which don't do anything). Once instrumentation is finished, the modified program is executed.

3.1 Limitations of Dynamic Binary Instrumentation

Although the advantages of dynamic binary instrumentation are manifold, there are some limitations that preclude us from fulfilling all the features or requirements of the POMP proposal. As we have had no experiences with binary instrumentation systems other than DPCL, it is not clear whether these are limitations of dynamic instrumentation in general or limitations of the DPCL instrumentation.

Events

Almost all of the events proposed by POMP can be instrumented. The exceptions are:

- Due to compiler optimization, the POMP events `Loop_iter_begin`, `Loop_iter_end`, and `Loop_iter_event` are not always visible at the binary level. For example, the compiler can choose to unroll (parts of) the parallel loop body. As it is not clear yet, whether the monitoring of these events really makes sense for OpenMP performance analysis, since they would introduce excessive overhead, we consider it to be acceptable for our tool not to handle these events.
- DPCL cannot instrument code inside shared libraries that are loaded into the executable at runtime. This would only be a problem if the code inside such a library uses OpenMP constructs (e.g., a mathematical parallel library). Unfortunately, the IBM OpenMP run time system is also linked as a shared library. Hence, the instrumentation of implicit barriers inside work-share constructs is not feasible. Possible work-arounds are to use static linking, which can be achieved with compiler flags, or to provide pre-instrumented versions of libraries, which use OpenMP. Unfortunately, both options would require re-linking of the executable.

Event Context Information

Not all the optional fields from the CTC string, as well as the run time context are obtainable at the binary level. This is the current status:

- Compile Time Context (CTC).
 - Available: region type, start SCL, end SCL, and function name
 - Not available: `hasFirstPrivate`, `hasLastPrivate`, `hasNowait`, `hasCopyin`, `schedule`, `hasOrdered`, and `hasCopypriv` attributes. We would need additional compiler support to provide these fields.
 - Available, but not implemented yet: `function_group`, and `hasReduction`.
- Run Time Context (RTC)
 - Available: `thread_ID`, and `num_threads`
 - Not available: `if_expr_result`
 - Available, but not implemented yet: `chunk_size`, `init_iter_value`, `iter_value`, `final_iter_value`, and `section_num`.

Open Issues in the POMP Specification

Unfortunately, the POMP specification (as described in [10]) is incomplete, i.e., not everything needed for a working implementation is specified. Here is a list of issues we came across:

- We had to define a format specification for the layout of the CTC strings describing the compile-time context of OpenMP events.

The string is a list of “attribute=value” pairs which are separated by a star (“*”) character (which is very unlikely to appear in function and file names). The string is prepended by the length of the string counting only the number of characters between the first star and the last, inclusive, without counting the C termination `\0` or any blank fill characters in Fortran. The string is terminated by an empty field, e.g., a double star “**”. The length is required, all other fields are optional. So the shortest possible string is “2**”.

We defined the attributes:

Attribute	Possible Values	Meaning
rtype	pre <code>region,loop,section,share,single,critical,master,ebarrier,ibARRIER,flush,atomic,ordered,func,user</code>	Region type
sscl	<code>filename[:lineI[:lineN]]</code>	Start source code location
escl	<code>[filename][:lineI[:lineN]]</code>	End source code location
name	<code>string</code>	Name of a function or name of a “named” critical region
group	<code>string</code>	For functions: Name of class, module, or namespace
schedule	<code>stat,dyn,guided,rt</code>	Schedule for parallel loops
hasfpriv	T,F	Boolean attributes of parallel regions and workshare constructs: absence of the attribute means false
haslpriv	T,F	
hasred	T,F	
hasnowait	T,F	
hascopyin	T,F	
hasord	T,F	
hascopypriv	T,F	

- It is not clear to what extent should user-defined functions be instrumented. We are currently instrumenting the begin and end of the main program and calls to user functions inside main but only outside of parallel regions. This seems to be a reasonable default that avoids excessive overhead.
- There needs to be an easy and compact way for a user to specify the extend of user function instrumentation.

3.2 Limitations of Source-code Instrumentation

For comparison, here is a list of limitations we determined when we tried to implement OPARI [12], a POMP instrumentation tool which works as a preprocessor (i.e., source-to-source translator).

- OPARI makes implicit barriers explicit. Unfortunately, this method cannot be used for measuring the barrier waiting time at the end of `PARALLEL` directives because they do not allow a `NOWAIT` clause. Therefore, OPARI adds an explicit barrier with corresponding performance interface calls here. For OPARI, this means that actually two barriers get called. But the second (implicit) barrier should execute and succeed immediately because the threads of the OpenMP team are already synchronized by the first barrier.

- The OpenMP standard (unfortunately) allows compilers to ignore `NOWAITs`, which means that in this case OPARI inserts an extra barrier and the POMP functions get invoked on this extra (and not the real) barrier.
- OPARI cannot instrument the internal synchronization inside `!$OMP WORKSHARE` as required by the OpenMP standard.
- We were told that some compilers use different implementations (with different characteristics) for implicit and explicit barriers. If OPARI changes implicit to explicit barriers, we measure the wrong behavior when using these compilers.

4 Examples of POMP Monitoring Libraries

To test and evaluate our dynamic instrumentation for POMP, we implemented three versions of POMP compliant monitoring libraries: a dummy library for overhead measurements, a POMP monitoring library for trace generation, and a POMP monitoring library for collection of profiling data.

4.1 Overhead of our POMP Library

The dummy POMP library implements all functionality necessary for all compliant implementations of the POMP interface without monitoring or measuring anything. Figure 3 shows the implementation of the `POMP_Parallel_begin()` event function. After checking whether POMP monitoring is activated at all and whether monitoring of the parallel begin event is activated, event context information is retrieved from the passed-in event handle. This version of the POMP library is used to measure the minimal overhead introduced by dynamic instrumentation for the POMP API. Our measurements on an IBM POWER4 system indicate that the minimum overhead for the instrumentation of one OpenMP or user function call is about 3 microseconds of cpu time.

```
int32 POMP_Parallel_begin(POMP_Handle_t handle, int32 thread_id) {
    if ( pomp_active && is_activated[MYPOMP_PAR_BEGIN] ) {
        mypompdescr* d = (mypompdescr*) handle;
        /* -- perform monitoring for parallel begin event here -- */
    }
    return 0;
}
```

Fig. 3. Typical implementation of a POMP event routine.

4.2 KOJAK POMP Library

We implemented a POMP monitoring library which generates EPILOG event traces. EPILOG is an open-source event trace format used by the KOJAK performance analysis tool framework [14]. Besides defining OpenMP related events, it provides a thread-safe implementation of the event reading, writing, and processing routines. In addition, it supports storing hardware counter and source code information and uses a (machine, node, process, thread) tuple to describe locations. This makes it especially well suited for monitoring Open or mixed MPI/OpenMP applications on today's clustered SMP architectures.

EPILOG event traces can either be processed by KOJAK's automatic event trace analyzer EXPERT or be converted to the VTF3 format used by the Vampir event trace visualization tool.

Figure 4 shows a screen-dump of the resulting display of the EXPERT automatic event trace analyzer. 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. We refer to [14] for a detailed description of KOJAK and EXPERT.

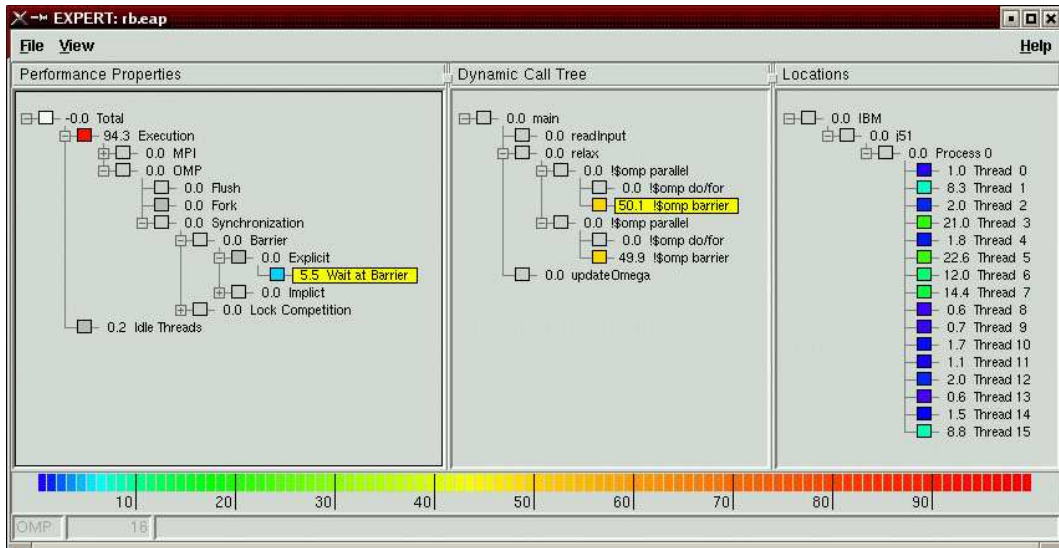


Fig. 4. Result display of EXPERT automatic trace analyzer.

Figure 5 shows (in the right half) a time line diagram of a small example application (shown in the source code display in the upper left corner). The EPILOG-to-VTF3 conversion maps OpenMP constructs into Vampir symbols and activities, as well as OpenMP barriers into a Vampir collective operation. This allows to see the dynamic behavior of an OpenMP application using a Vampir time-line diagram as well as to use Vampir's powerful filter and selection capabilities to generate all kind of execution statistics for any phase of the OpenMP application. In addition, all source code information contained in a trace is preserved during conversion; allowing the display of the corresponding source code simply by clicking on the desired activity.

4.3 POMP Profiler Library

The POMP Profiler library generates a detailed profile of the time spent in each parallel region of an application in the form of an XML file and in a text file. The XML file can be visualized by a graphical user interface, as shown in Figure 6. Total time taken by the master thread in a parallel event, the average time incurred by all work-sharing threads, and the deviation of the time spend by each from the average time are displayed for each of the tasks running in parallel. A detailed timing information of the threads is displayed by selecting the task of interest. The visualizer also maps the regions of the source code with the timing information. The timing information is crucial

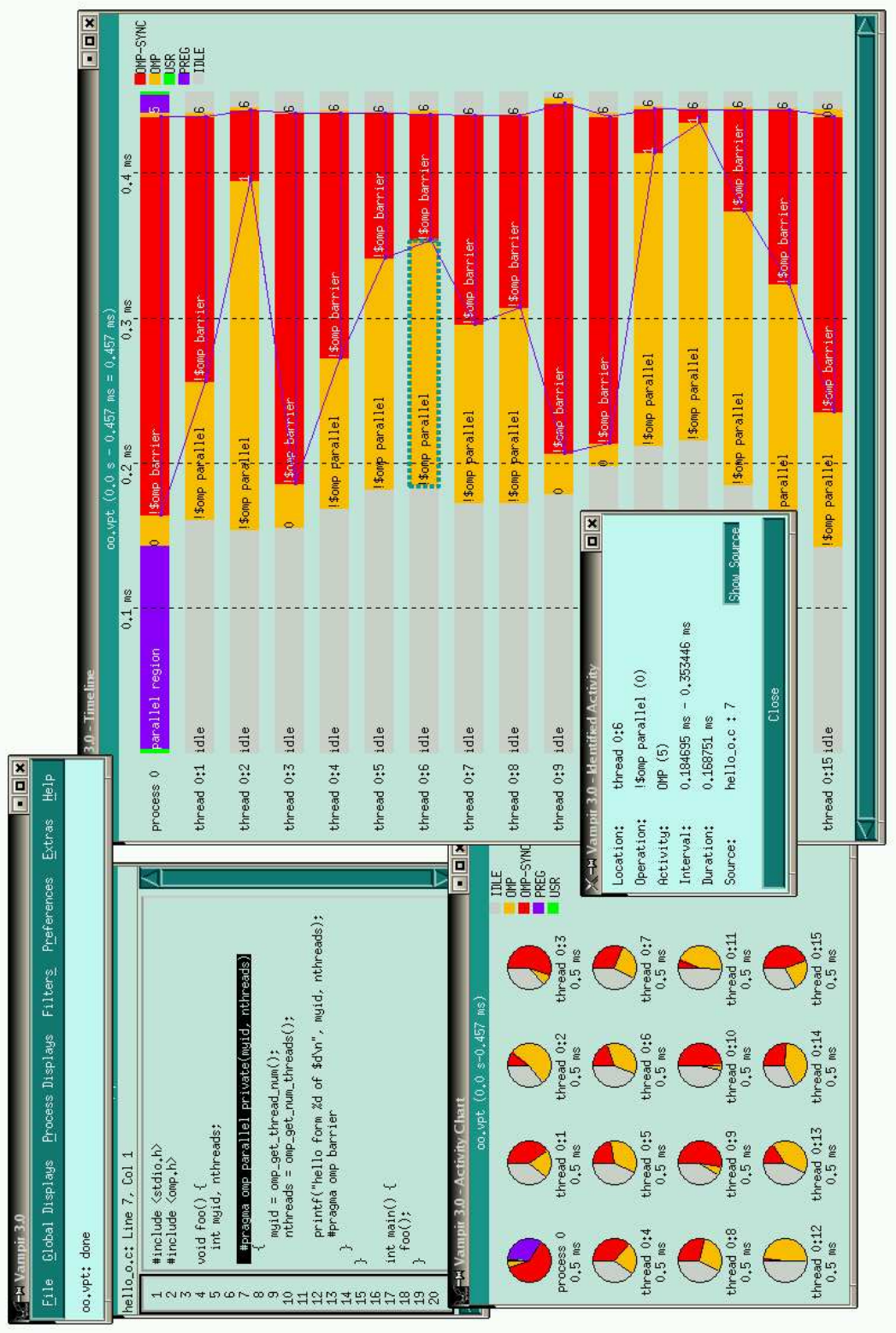


Fig. 5. Vampir time line diagram of example application.

in detecting load imbalances among threads and could also help application developers in choosing appropriate scheduling mechanism for parallel loops.

5 Conclusion

We presented an implementation of a performance monitoring interface for OpenMP based on the POMP proposal. Our POMP implementation uses binary instrumentation technology to insert calls to a POMP compliant library into the application binary for instrumentation of OpenMP directives, as well as user function calls. Our measurements, on an IBM POWER4 system, indicate that the instrumentation overhead is in the order of 3 microseconds of cpu time per call. In addition, we demonstrate the usability of our POMP interface with the interface of two versions of POMP compliant monitoring libraries: a library for trace generation and a library for collection of profiling data.

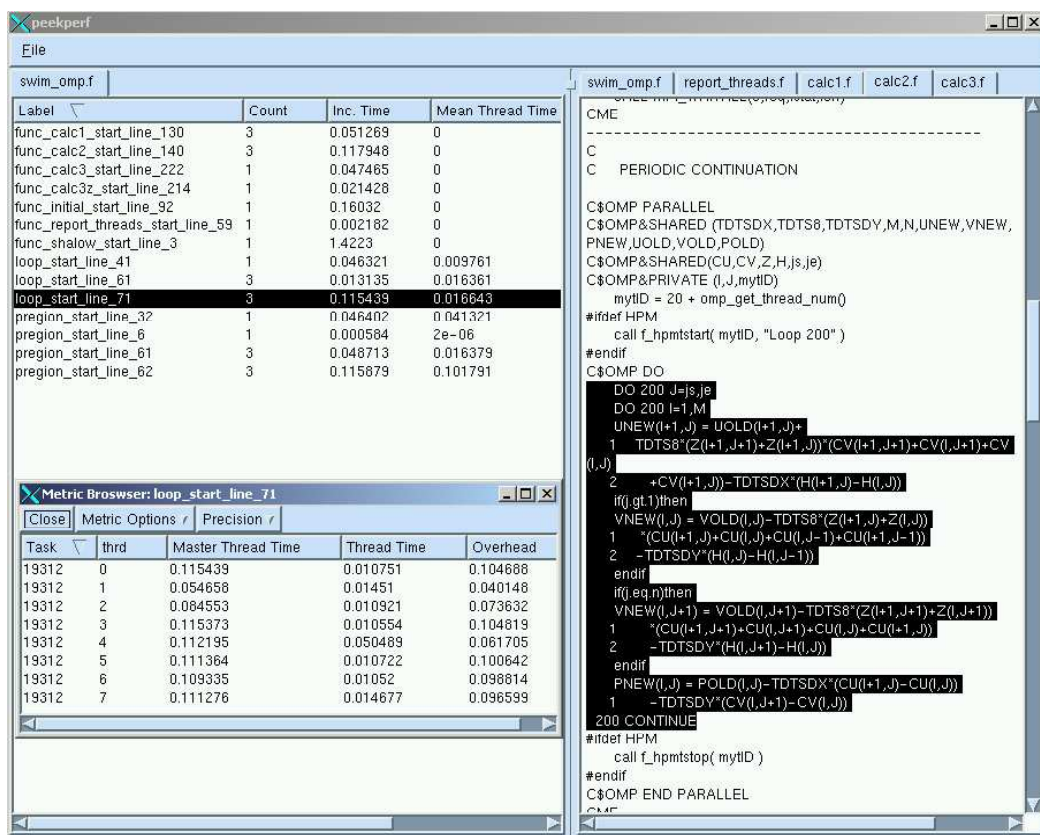


Fig. 6. Visualization of data generated by Profiler POMP Library for an example application.

References

1. E. Ayguadé, M. Brorsson, H. Brunst, H.-C. Hoppe, S. Karlsson, X. Martorell, W. E. Nagel, F. Schlimbach, G. Utrera, and M. Winkler. OpenMP Performance Analysis Approach in the INTONE Project. In *Proceedings of the Third European Workshop on OpenMP - EWOMP'01*, September 2001.

2. B. R. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.
3. Jordi Caubet, Judit Gimenez, Jesús Labarta, Luiz DeRose, and Jeffrey Vetter. A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications. In *Proceedings of the Workshop on OpenMP Applications and Tools - WOMPAT 2001*, pages 53 – 67, July 2001.
4. Luiz DeRose. The Hardware Performance Monitor Toolkit. In *Proceedings of Euro-Par*, pages 122–131, August 2001.
5. Luiz DeRose, Ted Hoover Jr., and Jeffrey K. Hollingsworth. The Dynamic Probe Class Library - An Infrastructure for Developing Instrumentation for Performance Tools. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2001.
6. Luiz DeRose and Daniel Reed. Svpablo: A Multi-Language Architecture-Independent Performance Analysis System. In *Proceedings of the International Conference on Parallel Processing*, pages 311–318, August 1999.
7. Luiz DeRose and Felix Wolf. CATCH - A Call-Graph Based Automatic Tool for Capture of Hardware Performance Metrics for MPI and OpenMP Applications. In *Proceedings of Euro-Par*, August 2002.
8. Seon Wook Kim, Bob Kuhn, Michael Voss, Hans-Christian Hoppe, and Wolfgang Nagel. VGV: Supporting Performance Analysis of Object-Oriented Mixed MPI/OpenMP Parallel Applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2002.
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 Paradyne Parallel Performance Measurement Tools. *IEEE Computer*, 28(11):37–46, November 1995.
10. B. Mohr, A. Mallony, H-C. Hoppe, F. Schlimbach, G. Haab, and S. Shah. A Performance Monitoring Interface for OpenMP. In *Proceedings of the fourth European Workshop on OpenMP - EWOMP'02*, September 2002.
11. Bernd Mohr, Allen Malony, and Janice Cuny. TAU Tuning and Analysis Utilities for Portable Parallel Programming. In G. Wilson, editor, *Parallel Programming using C++*. M.I.T. Press, 1996.
12. Bernd Mohr, Allen Malony, Sameer Shende, and Felix Wolf. Towards a Performance Tool Interface for OpenMP: An Approach Based on Directive Rewriting. In *Proceedings of the Third European Workshop on OpenMP - EWOMP'01*, September 2001.
13. Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In Anthony Skjellum, editor, *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society, 1993.
14. 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*, February 2003.