

# Profiling and Tracing OpenMP Applications with POMP Based Monitoring Libraries

Luiz DeRose<sup>1</sup>, Bernd Mohr<sup>2</sup>, and Seetharami Seelam<sup>3</sup>

<sup>1</sup> Cray Inc.

Mendota Heights, MN, USA

ldr@cray.com

<sup>2</sup> Forschungszentrum Jülich, ZAM,

Jülich, Germany

b.mohr@fz-juelich.de

<sup>3</sup> University of Texas at El Paso

El Paso, TX, USA

seelam@cs.utep.edu

**Abstract.** In this paper we present a collection of tools that are based on the POMP performance monitoring interface for analysis of OpenMP applications. These POMP compliant libraries, POMPROF and the KOJAK POMP library, provide respectively the functionality for profiling and tracing of OpenMP applications. In addition, we describe a new approach to compute temporal overhead due to scheduling (load-imbalance), synchronization (barrier time), and the runtime system. Finally, we exemplify the use of these libraries with performance measurement and visualization of the ASCI SPPM benchmark code. Our examples show that the information provided by both tools is consistent, and provides data that is helpful for users to understand the source of performance problems.

## 1 Introduction

OpenMP is today's de facto standard for shared memory parallel programming of scientific applications. It provides a higher level specification for users to write threaded programs that are portable across most shared memory multiprocessors. However, application developers still face application performance problems such as load imbalance and excessive barriers overhead, when using OpenMP. Moreover, these problems are difficult to detect without the help of performance tools.

The MPI specification defines a standard monitoring interface (PMPI), which facilitates the development of performance monitoring tools. Similarly, a Java Virtual Machine Profiler Interface (JVMPi) is an experimental feature in the Java 2 SDK, which is intended for users and tools vendors to develop profilers that work in conjunction with the Java virtual machine implementation. Unfortunately, OpenMP does not provide yet a standardized performance monitoring interface, which would simplify the design and implementation of portable OpenMP performance tools. Mohr et. al. [9] proposed POMP, a performance monitoring interface for OpenMP. This proposal extends experiences of previous implementations of monitoring interfaces for OpenMP [1, 7, 10]. POMP describes an API to be called by “*probes*” inserted into the application by a

compiler, a pre-processor, or via a binary or dynamic instrumentation mechanism. With such performance monitoring interface, users and tools builders can then define their own POMP compliant libraries for performance measurement of OpenMP applications.

In [6], we presented DPOMP, a POMP instrumentation tool based on binary modification, which takes as input a performance monitoring library that conforms to the POMP API and an OpenMP application binary. DPOMP instruments the binary of the application with dynamic probes containing POMP calls defined in the library. In this paper we present two POMP compliant libraries, POMPROF and the KOJAK POMP library, which provide respectively the functionality for profiling and tracing of OpenMP applications. In addition, we exemplify the use of these libraries with performance measurement and visualization of the ASCI sPPM benchmark code.

The remainder of this paper is organized as follows: In Section 2 we briefly describe the main features of DPOMP. In Section 3 we describe our POMP compliant library for profiling of OpenMP applications. In Section 4 we describe the KOJAK POMP library for tracing of OpenMP programs. In Section 5 we present examples of utilization of these two libraries. Finally, we present our conclusions in Section 6.

## 2 A POMP Instrumentation Tool Based on Binary Modification

DPOMP was implemented using DPCL [5], an object-based C++ class library and runtime infrastructure based on the Dyninst Application Programming Interface (API) [3]. Using DPCL, a performance tool can insert code patches at function entry and exit points, as well as before and after call sites. Since the IBM compiler translates OpenMP constructs into functions that call “*outlined*” functions containing the body of the construct, DPOMP can insert calls to functions of a POMP compliant monitoring library for each OpenMP construct in the target application<sup>1</sup>. The main advantage of this approach lies in its ability to modify the binary with performance instrumentation with no special preparations, like re-compiling or re-linking. In addition, since it relies only on the binary, DPOMP works independently of the programming language used for the OpenMP program.

DPOMP takes as input an OpenMP application binary and a POMP compliant performance monitoring library. It reads the application binary, as well as the binary of the POMP library and instruments the application binary, so that, at locations which represent events in the POMP execution model the corresponding POMP monitoring routines are called. From the user’s point of view, 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. From the tools builder point of view, instrumentation can also be controlled by the set of POMP routines provided by the library, i.e., instrumentation is only applied to those events that have a corresponding POMP routine in the 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). By

<sup>1</sup> Notice that DPOMP has a dependence on the naming convention used by the IBM XL compiler. However, it can be easily extended to any other OpenMP compiler that uses the “function outline” approach.

default, DPOMP instruments all OpenMP constructs for which there is a corresponding POMP function in the library. It also instruments all user functions called from the main program provided that there is a definition in the library for performance monitoring of user functions. In addition, for MPI applications, by default DPOMP instruments all MPI calls in the program. Once instrumentation is finished, the modified program is executed.

### 3 The POMP Profiler Library

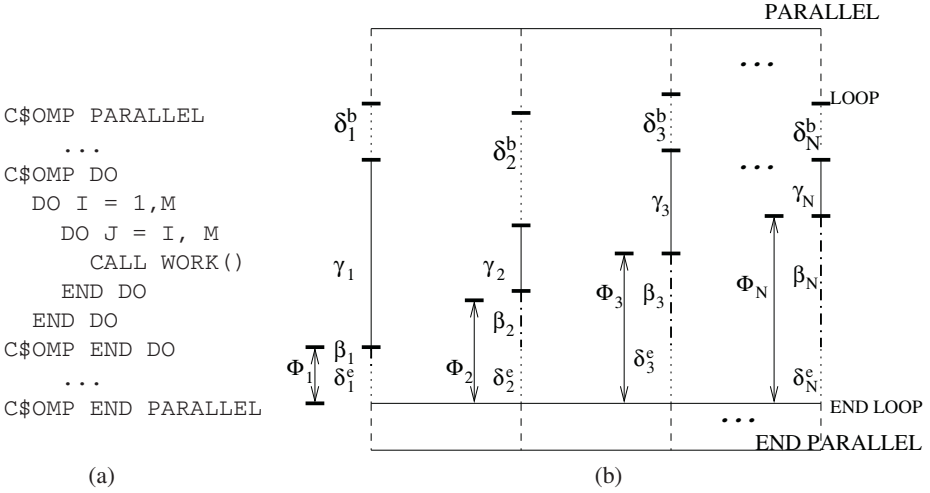
We implemented a POMP compliant monitoring library (POMPPOF) that generates a detailed profile describing various overheads and the amount of time spent by each thread in three key regions of the parallel application: parallel regions, OpenMP loops inside a parallel region, and user defined functions. In addition, POMPPOF provides information for analysis of the slowdown caused by sequential regions, as well as measurements of execution time from user functions and OpenMP API calls such as `set lock`. The profile data is presented in the form of an XML file that can be visualized by a graphical interface as shown in Figure 3.

POMPPOF performs overhead analysis, which is a key incremental method for performance tuning of parallel applications [4]. Historically, both temporal and spatial overheads have been defined as a function of two execution times of an application: namely, the execution time of the sequential code, and the execution time of the parallel code [4, 12]. One of the problems of this approach is that it requires an application programmer to have two versions of a program, a serial and a parallel. Moreover, often overhead definitions for OpenMP do not include the property of measurability of basic blocks of code, reason being that it is not always possible to measure all overhead sources at the OpenMP construct level.

Since we can use our binary instrumentation infrastructure to monitor blocks of OpenMP code, which are packaged into outlined functions, we compute temporal overheads with measurable times. Measurable temporal overhead often results from an imbalanced loop, a barrier for synchronization, or from the runtime code. So, POMPPOF focuses on temporal overhead that is due to scheduling (load-imbalance), synchronization (barrier time), and the overhead due to the runtime system.

The main goal of our profiler library is to help application programmers tune their parallel codes, by presenting a “thread-centered” temporal overhead information. Hence, we define temporal overhead in terms of the total amount of time in the OpenMP constructs by each of the “ $N$ ” threads in a parallel region. We assume that each of the “ $N$ ” threads runs on a different processor. To illustrate our definition and measurement of the three components of overhead, we use a simple triangular loop inside an OpenMP parallel region with an implicit barrier at the end, shown in Figure 1(a).

The code in Figure 1(a) leads to an execution pattern similar to the one depicted in Figure 1(b). Let us assume that  $N$  threads are created by the parallel region. We define for each thread  $i$ , where  $(1 \leq i \leq N)$ ,  $\delta_i^b$  as the runtime system overhead before the thread begins executing the work;  $\delta_i^e$  as the runtime system overhead after the thread ends the execution of the work;  $\beta_i$  as the time spent on barrier synchronization;  $\gamma_i$  as the time spent executing the allocated work; and finally  $\phi_i$  as the exit overhead,



**Fig. 1.** (a) Pseudo code for a triangular loop inside an OpenMP parallel region, and (b) possible execution pattern

which is the sum of the barrier overhead and the exit overhead of the runtime system, i.e.,  $\phi_i = \beta_i + \delta_i^e$ . The total execution time ( $\tau_i$ ) of each thread  $i$  is the sum of these components, i.e.,  $\tau_i = \delta_i^b + \delta_i^e + \beta_i + \gamma_i$ .

**Load imbalance** basically occurs because of un-equal amount of work is being distributed to the threads. In [2] load imbalance is defined as the difference between the time taken by the slowest thread and the mean thread time, which essentially is a constant for all threads. As we can see from Figure 1, each thread  $i$  may have a different amount of execution time  $\gamma_i \neq 0$  leading to different amounts of load imbalance. Hence we need a thread-centered definition of load imbalance.

For performance enhancement reasons (e.g., selecting an appropriate scheduling technique), in our approach to define load imbalance we focus on how much worse each thread  $i$  is performing with respect to the thread  $j$  that takes the minimum amount of time. Hence our definition of load imbalance is the percentage of extra time spent by each thread in computation normalized with respect to the time of the fastest thread. Thus, the load imbalance of each thread  $i$ , expressed in percentage, is computed as

$$l_i = \frac{(\gamma_i - \min(\gamma_i))}{\min(\gamma_i)} \times 100.$$

**Barrier overhead** is the amount of time spent by each thread  $i$  while waiting for synchronization with other threads, after its work share is completed. In OpenMP there are two types of barriers: explicit and implicit. Explicit barriers are often visible at the binary level; hence, they can be instrumented and measured for each thread. On the other hand, implicit barriers, which are normally used by OpenMP loop constructs, such as the one shown in our example, are executed inside of the runtime library, and are not visible in the application binary. Hence, they cannot be instrumented with DPOMP [6].

In order to estimate the implicit barrier overhead ( $\beta_i$ ) for implicit barriers, we use the measured total exit-overhead  $\phi_i$ , as follows:

We consider that the last thread to join the barrier (say thread  $j$ ) incurs the least amount of exit overhead and has zero barrier time. Hence, since  $\phi_j = \beta_j + \delta_j^e$ , with  $\beta_j = 0$ , we can assume that the runtime exit overhead  $\delta_j^e = \phi_j$ . The runtime exit system overhead  $\delta_i^e$  of each thread  $i$  is a constant  $\varepsilon$  for all practical purposes, because all threads executes the same runtime system code. Hence, we can compute the barrier time of each thread  $i$  as  $\beta_i = \phi_i - \varepsilon$ , where  $\varepsilon = \min(\delta_i^e)$  for  $(1 \leq i \leq N)$ .

In case of a “NO WAIT” clause at the end of the loop, the threads incur only an exit runtime overhead, and the barrier overhead is considered zero.

**Runtime overhead** is the amount of time taken by each thread  $i$  to execute the runtime system code. For each thread  $i$ , the runtime overhead  $\delta_i$  is computed as the sum of the overheads at the beginning of the loop ( $\delta_i^b$ ) and at the end of the loop ( $\delta_i^e$ ), i.e.,  $\delta_i = \delta_i^b + \delta_i^e$ . In case of an explicit barrier at the end of the loop, both  $\delta_i^b$  and  $\delta_i^e$  can be instrumented and measured. However, for implicit barriers, only  $\delta_i^b$  is measured, while  $\delta_i^e$  is estimated as described in the barrier overhead.

## 4 The 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 [13]. 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 OpenMP 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 commercial Vampir event trace visualization tool [11] (not shown here due to space limitation).

Figure 2 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. The severity is expressed in percentage of execution time lost due to this problem. By expanding or collapsing nodes in each of the three trees, the analysis can be performed on different levels of granularity. We refer to [13] for a detailed description of KOJAK and EXPERT.

If a more detailed (manual) analysis is needed, EPILOG traces can be converted to VTF3 format suitable for Vampir. The conversion maps OpenMP constructs into Vampir symbols and activities, as well as OpenMP barriers into a Vampir collective operation. This allows users to investigate 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

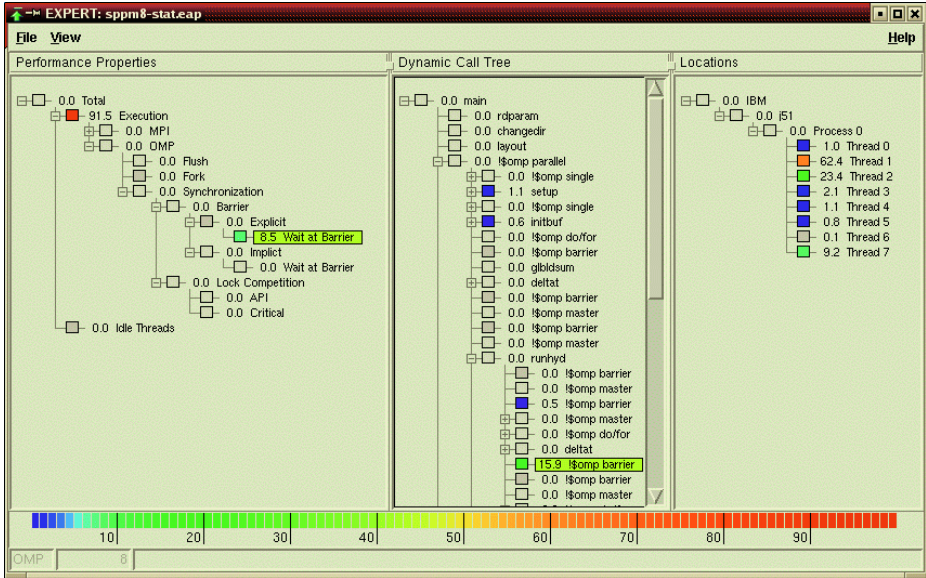


Fig. 2. Result display of EXPERT automatic trace analyzer.

during conversion; allowing the display of the corresponding source code simply by clicking on the desired activity.

## 5 Examples of Use

In this section we exemplify the use of both libraries with performance measurements and visualization of the ASCII SPPM benchmark code [8], which solves a 3D gas dynamics problem on a uniform cartesian mesh using a simplified version of the Piecewise parabolic method, with nearest neighbor-communication. The SPPM benchmark is a hybrid code (OpenMP and MPI), written in Fortran 77 with some C routines. Figure 3 shows the summary view for the program and the detailed view for one of the loops (loop 1125) from the profile data obtained with POMPROF, when using “static” scheduling for the OpenMP loops, running 8 threads on an IBM p690+. In the summary view, which displays the highest value for each metric, we observe very high values for “% Imbalance” on all the loops. The detailed view for the loop in line 1125 confirms this imbalance. When replacing the static scheduling by dynamic scheduling, we observe a much better behavior of the code with respect to “% Imbalance”, as shown in Figure 4.

These findings are also confirmed by the automatic trace analysis of EXPERT. As shown in Figure 2, the imbalance caused by the static scheduling of OpenMP loop iterations results in a total of 8.5% of waiting time at explicit barriers. For the selected barrier, the uneven distribution of this waiting time can be seen in the right pane. For dynamic scheduling (not shown here), waiting time only amounts to 0.5%. By selecting the property “Execution” in the left pane and one of the “!\$omp do/for” loops in the middle pane, the right pane of EXPERT would show the (im)balance in execution time.

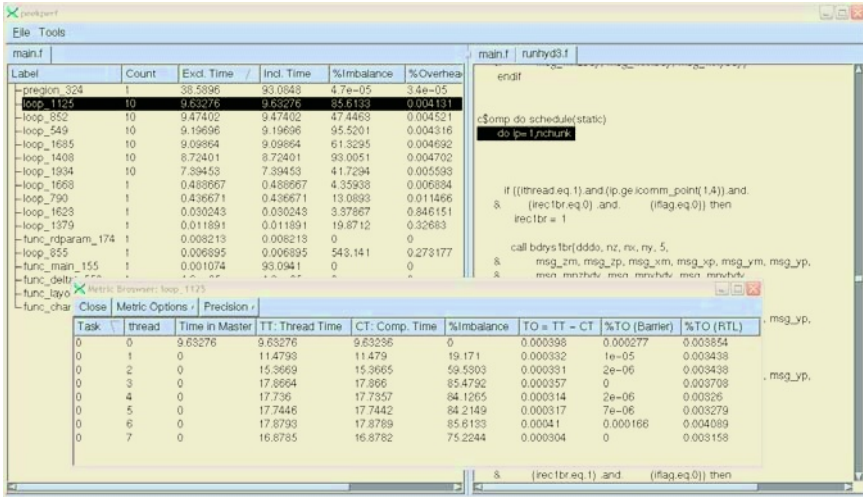


Fig. 3. Visualization of the POMPROF data from SPPM and detailed view of Loop 1125 implemented with static scheduling



Fig. 4. Detailed view of Loop 1125, modified to use dynamic scheduling

## 6 Conclusion

We presented a collection of tools for the analysis of OpenMP applications based on the DPOMP instrumentation infrastructure. The use of a standard monitoring interface like POMP allows the utilization of a variety of measurement methods and provides the flexibility for development of tools ranging from profilers to tracers for performance analysis, debugging, and tuning. We presented the POMPROF profiling library and the KOJAK performance analysis framework which not only includes the EXPERT automatic trace analyzer but is also integrated with the Vampir trace visualization system. Other tools can be easily integrated or developed from scratch by implementing a POMP compliant monitoring library. In addition, we exemplified the use of these libraries with performance measurement and visualization of the ASCII SPPM benchmark code. Through the use of appropriate PMPI wrapper libraries which, if needed, record MPI point-to-point and collective communication performance data, the DPOMP infrastructure can also be used to monitor hybrid OpenMP / MPI applications. However, this requires an extra re-linking step before execution. We are currently working on extending DPOMP to avoid this problem.

## 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. M. K. Bane and G. D. Riley. Automatic overheads profilers for openmp codes. In *Second European Workshop on OpenMP (EWOMP 2000)*, Edinburgh, Scotland, September 2000.
3. 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.
4. J. M. Bull. A hierarchical classification of overheads in parallel programs. In *First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*, Chapman Hall, pages 208–219, 1996.
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, Bernd Mohr, and Seetharami Seelam. An Implementation of the POMP Performance Monitoring Interface for OpenMP Based on Dynamic Probes. In *Proceedings of the fifth European Workshop on OpenMP - EWOMP'03*, September 2003.
7. 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.
8. Lawrence Livermore National Laboratory. *the sPPM Benchmark Code*, 2002. <http://www.llnl.gov/asci/purple/benchmarks/limited/sppm/>.
9. 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.
10. 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.
11. W. Nagel, A. Arnold, M. Weber, H-C. Hoppe, and K. Solchenbach. Vampir: Visualization and Analysis of MPI Resources. *Supercomputer*, 12:69–80, January 1996.
12. G. D. Riley, J. M. Bull, and J. R. Gurd. Performance improvement through overhead analysis: A case study in molecular dynamics. In *International Conference on Supercomputing*, pages 36–43, 1997.
13. Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid mpi/openmp applications. *Journal of Systems Architecture, Special Issue 'Evolutions in parallel distributed and network-based processing'*, 49(10–11):421–439, November 2003.