ompP: A Profiling Tool for OpenMP *

Karl Fürlinger and Michael Gerndt

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

Abstract. In this paper we present a simple but useful profiling tool for OpenMP applications similar in spirit to the MPI profiler `mpiP` [15]. We describe the implementation of our tool and demonstrate its functionality on a number of test applications.

1 Introduction

For developers of scientific and commercial applications it is essential to understand the performance characteristics of their codes in order to take most advantage of the available computing resources. This is especially true for parallel programs, where a programmer additionally has to take issues such as load balancing, synchronization and communication into consideration. Accordingly, a number of tools with varying complexity and power have been developed for the major parallel programming languages and systems.

Generally, tools collect performance data either in the form of traces or profiles. Tracing allows a more detailed analysis as temporal characteristics of the execution is preserved, but it is usually more intrusive and the analysis of the recorded traces can be involved and time-consuming. Profiling, on the other hand, has the advantage of giving a concise overview where time is spent while causing less intrusion.

The best-known tracing solution for MPI is Vampir [12] (now Intel Trace Analyzer [6]) while `mpiP` [15] is a compact and easy to use MPI profiler. Both Vampir and `mpiP` rely on the MPI profiling interface that allows the interception and replacement of MPI routines by simply re-linking the user-application with the tracing or profiling library. Unfortunately no similar standardized profiling or performance analysis interface exists for OpenMP yet, making OpenMP performance analysis dependant on platform- and compiler specific mechanisms.

Fortunately, a proposal for a profiling interface for OpenMP is available in the form of the POMP specification and an instrumenter called Opari [10] has been developed that inserts POMP calls around instrumented OpenMP constructs. The authors of POMP and Opari also provide a tracing library, while we have

* This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG) under contract GE1635/1-1.

implemented a straightforward POMP-based profiler that is similar in spirit to `mpiP` and which accordingly we call `ompP` [**?**].

The rest of the paper is organized as follows: In Sect. 2 we describe the design and implementation of our tool and in Sect. 3 we demonstrate its functionality on some example programs. Finally in Sect. 4 we review related work, we conclude and present ideas for future work in Sect. 5.

2 Tool Design and Implementation

In this section we present the design and implementation of our profiling tool `ompP`.

2.1 Instrumentation

Opari [10] is an OpenMP source-to-source instrumenter for C, C++ and Fortran developed by Mohr et al. that inserts calls to a POMP compliant monitoring library around OpenMP constructs. For each instrumented OpenMP construct Opari creates a *region descriptor* structure that contains information such as the name of the construct, the source file and the begin and end line numbers. Each `POMP_*` call passes a pointer to the descriptor of the region being affected. In the example shown in Fig. 1, Opari creates one region descriptor for the parallel region and this descriptor is used for the `POMP_Parallel_[fork,join,begin,end]` and also for the `POMP_Barrier_[Enter,Exit]` calls. The barrier is added by Opari in order to measure the load imbalance in the parallel region, similar *implicit* barriers are added to OpenMP worksharing constructs.

```
POMP_Parallel_fork [master]          ] enter          ] main
#pragma omp parallel {
    POMP_Parallel_begin [team]       ]        ] body  |
                                              |       |
        POMP_Barrier_Enter [team]    ] ibarr  |       |
        #pragma omp barrier                   |       |
        POMP_Barrier_Exit [team]     ]        |       |
    POMP_Parallel_end [team]         ] exit   ]       |
}                                             |       |
POMP_Parallel_join [master]          ]                ]
```

Fig. 1. Instrumentation added by Opari for the OpenMP `parallel` construct. The original code is shown in boldface, the square brackets denote the threads that execute a particular `POMP_*` call. The right part shows the pseudo region nesting used by `ompP`.

2.2 Performance Data Collection

Our profiler keeps track of counts and inclusive times for the instrumented OpenMP constructs. In order to simplify performance data bookkeeping (the same region descriptor can be used in a multitude of `POMP_*` calls), each Opari region is broken down into smaller conceptual "pseudo" regions and performance data (i.e, timestamps and execution counts) are recorded on the basis of these pseudo regions. In the example shown in Fig. 1, the pseudo regions are `main`, `body`, `enter`, `exit` and `ibarr`.

Instead of keeping track of all possible `POMP_*` calls for the individual Opari regions there are only two events for a pseudo region, namely `enter` and `exit`. For an `enter` event we record the enter timestamp (wall-clock time) that is later used in the `exit` event to increment the summed execution time by the elapsed time. Additionally a counter is incremented to count the number of executed instances of the pseudo region.

	seq	main	body	ibarr	enter	exit
MASTER	×					
ATOMIC		×				
BARRIER		×				
FLUSH		×				
USER_REGION		×				
LOOP		×		×		
SECTIONS		×	×	×		
SINGLE		×	×	×		
CRITICAL		×	×		×	×
WORKSHARE		×		×		
PARALLEL	×		×	×	×	×
PARALLEL_LOOP	×		×	×	×	×
PARALLEL_SECTIONS	×	×	×	×	×	×
PARALLEL_WORKSHARE	×		×	×	×	×

Fig. 2. List of pseudo regions for the different OpenMP constructs.

A list of pseudo regions for the different Opari regions is shown in Fig. 2, the first column gives the name of the corresponding OpenMP construct as reported by `ompP` (`LOOP` refers to the `for` construct in C and the `do` construct in Fortran).

The pseudo regions have the following semantic meaning:

`main` Corresponds to the main region of the construct (unless the region is executed by one thread only then this role is taken by `seq`), if the construct has nested sub-regions, this refers to the "outer" part of a construct. An example is a `sections` construct that contains one or more `section` blocks.

`body` Corresponds to the "inner" part of a construct, for example a `section` region inside a `sections` directive.

`ibarr` Corresponds to the implicit barrier added by Opari to worksharing constructs (unless a `nowait` clause is present) to measure load imbalance.

`enter` Allows the measurement of the time required to enter a construct. For critical sections this is the waiting time at the entry of the critical section. For parallel sections (and combined parallel worksharing regions) this measures the thread startup overhead.

`exit` Measures the time required to leave a construct. For critical sections this is the time required for leaving the critical section.[1] For parallel sections and combined parallel worksharing constructs this corresponds to the thread teardown overhead.

`seq` Measures the sequential execution time for a construct, i.e., the time spent by the master thread in a `master` construct or the time passing between `POMP_Fork` and `POMP_Join` in a `parallel` construct or combined parallel worksharing constructs.

Performance data is collected on a *region stack* basis. That is, similar to a call-path profile [8, 3] where performance data is attributed not to a function itself (that would be a flat profile) but rather to the call-path that leads to a function, a stack of entered Opari regions is maintained and data is attributed to the stack that leads to a certain region. This region stack is currently maintained for POMP regions only, i.e., only automatically instrumented `OpenMP` constructs and user-instrumented regions[2] are placed on the stack, general functions or procedures are not handled unless manually instrumented by the user.

Including all called functions in our region stack would certainly be useful. However, this requires us to either perform a stackwalk (as `mpiP` does) or make use of compiler-supplied function instrumentation (i.e., the `-f instrument-functions` for the GNU compiler collection). Note that in either approach unwanted exposition to the compiler's implementation of the OpenMP standard (e.g., compiler outlining of parallel regions) has to be expected.

2.3 Performance Data Presentation

The performance data collected by `ompP` is kept in memory and written to a report file when the program finishes. The report file has the following sections:

- A header containing general information such as date and time of the program run.
- A list of all identified Opari regions with their type (`PARALLEL`, `ATOMIC`, `BARRIER`, ...) source file and line number information.
- A region summary list where performance data is summarized over the threads in the parallel execution. This list is sorted according to summed execution time and is intended to enable the developer to quickly identify the most time-consuming regions (and thus the most promising optimization targets).

[1] Usually one doesn't expect much waiting time at the end of a critical section. However, a thread might incur some overhead for signaling the critical section as "free" to other waiting threads.

[2] Users can instrument arbitrary regions by using the `pomp inst begin(`*name*`)` and `pomp inst end(`*name*`)` pragmas.

– A detailed region summary for each identified region and for a specific region stack. This information allows the identification of load imbalances in the execution time and many other causes of inefficient or incorrect behavior.
– A region summary for each region, where data is summed over all different region stacks that lead to the particular region (i.e., the flat profile for the region).

To produce a useful and concise profiling report, data are not reported as times and counts for each individual pseudo regions but specific semantic names are given according to the underlying Opari region. The following times and counts are reported:

– `execT` and `execC` count the number of executions and the total inclusive time spent for each thread (this is derived from the `main` or `body` pseudo region depending on the particular OpenMP construct).
– `exitBarT` and `exitBarC` are derived from the `ibarr` pseudo region and correspond to time spent in the implicit "exit barrier" in worksharing constructs or parallel regions. Analyzing the distribution of this time reveals load imbalances.
– `startupT` and `startupC` are defined for the OpenMP `parallel` construct and for the combined parallel work-sharing constructs (`parallel for` and `parallel sections` and `parallel workshare`), the data is derived from the `enter` pseudo region. If large fraction of time is spent in `startupT` and `startupC` is high, this indicates that a parallel region was repeatedly executed (maybe inside a loop) causing high overhead for thread creation and destruction.
– `shutdownT` and `shutdownC` are defined for the OpenMP `parallel` construct and for the combined parallel work-sharing constructs, the data is derived from the `exit` pseudo region. Its interpretation is similar to `startupT` and `startupC`.
– `singleBodyT` and `singleBodyC` are reported for `single` regions and report the time and execution counts spent inside the single region for each thread, the data is derived from the `body` pseudo region.
– `sectionT` and `sectionC` are reported for a `sections` construct and give the time and counts spent inside a `section` construct for each thread. The data is derived from the `body` pseudo region.
– `enterT`, `enterC`, `exitT` and `exitC` give the counts and times for entering and exiting `critical` sections, the data is derived from the `enter` and `exit` pseudo regions.

3 Application Examples

We report on a number of experiments that we have performed with `ompP`, all measurements have been performed on a single 4-way Itanium-2 SMP systems (1.3 GHz, 3 MB third level cache and 8 GB main memory), the Intel compiler version 8.0 was used.

3.1 APART Test Suite (ATS)

The ATS [11] is a set of test applications (MPI and OpenMP) developed within the APART [3] working group to test the functionality of automated and manual performance analysis tools. The framework is based on functions that generate a sequential amount of work for a process or thread and on a specification of the distribution of work among processes or threads. Building on this basis, individual programs are generated that exhibit a certain pattern of inefficient behavior, for example "imbalance in parallel region".

Previous work already tested existing OpenMP performance analysis tools with respect to their ability to detect the performance problems in the ATS framework [2]. With Expert [16], also a POMP-based tool was tested and generally with `ompP` a developer is able to detect the same set of OpenMP related problems as Expert (although with Expert the process is somewhat more automated).

The `ompP` output below is from a profiling run for the ATS program that demonstrates the "imbalance in parallel loop" performance problem. Notice the `exitBarT` column and the uneven distribution of time with respect to threads {0,1} and {2,3}. This example is typical for a number of load imbalance problems that are easily spottable by analyzing the exit barrier.

```
R00003   LOOP            pattern.omp.imbalance_in_parallel_loop.c (15--18)
   001:  [R0001]  imbalance_in_parallel_loop.c (17--34)
   002:  [R0002]  pattern.omp.imbalance_in_parallel_loop.c (11--20)
   003:  [R0003]  pattern.omp.imbalance_in_parallel_loop.c (15--18)

 TID     execT     execC  exitBarT  exitBarC
  0       6.32         1      2.03         1
  1       6.32         1      2.02         1
  2       6.32         1      0.00         1
  3       6.32         1      0.00         1
  *      25.29         4      4.05         4
```

3.2 Quicksort

Süß and Leopold compare several parallel implementations of the Quicksort algorithm with respect to their efficiency in representing its recursive divide-and-conquer nature [14]. The code is now part of the OpenMP source code repository [1] and we have analyzed a version with a global work stack (called `sort_omp_1.0` in [14]) with `ompP`. In this version there is a single stack of work elements (sub-sequences of the vector to be sorted) that are placed on or taken from the stack by the threads. Access to the stack is protected by critical section. The `ompP` output below shows the two critical sections in the code and it clearly indicates that a considerable amount of time is spent due to critical section contention. The total execution time of the program (summed over threads) was 61.02 seconds so the 9.53 and 6.27 seconds represent a considerable amount.

[3] Automated Performance Analysis: Real Tools

```
R00002   CRITICAL        cpp_qsomp1.cpp (156--177)
   001:  [R0001]  cpp_qsomp1.cpp (307--321)
   002:  [R0002]  cpp_qsomp1.cpp (156--177)
 TID     execT     execC    enterT    enterC     exitT     exitC
  0       1.61    251780      0.87    251780      0.31    251780
  1       2.79    404056      1.54    404056      0.54    404056
  2       2.57    388107      1.38    388107      0.51    388107
  3       2.56    362630      1.39    362630      0.49    362630
  *       9.53   1406573      5.17   1406573      1.84   1406573

R00003   CRITICAL        cpp_qsomp1.cpp (211--215)
   001:  [R0001]  cpp_qsomp1.cpp (307--321)
   002:  [R0003]  cpp_qsomp1.cpp (211--215)
 TID     execT     execC    enterT    enterC     exitT     exitC
  0       1.60    251863      0.85    251863      0.32    251863
  1       1.57    247820      0.83    247820      0.31    247820
  2       1.55    229011      0.81    229011      0.31    229011
  3       1.56    242587      0.81    242587      0.31    242587
  *       6.27    971281      3.31    971281      1.25    971281
```

To improve the performance of the code, Süß and Leopold implemented a second version using thread-local stacks to reduce the contention for the global stack. We also analyzed the second version with `ompP` and the timing result for the two critical sections appears below.

In this version the overhead with respect to critical sections is clearly smaller than the first one (`enterT` and `exitT` have been improved by about 25 percent) The overall summed runtime reduces to 53.44 seconds, an improvement of about 12 percent, which is in line with the results reported in [14]. While this result demonstrates a nice performance gain with relatively little effort, our analysis clearly indicates room for further improvement; an idea would be to use lock-free data structures.

```
R00002   CRITICAL        cpp_qsomp2.cpp (175--196)
   001:  [R0001]  cpp_qsomp2.cpp (342--358)
   002:  [R0002]  cpp_qsomp2.cpp (175--196)
 TID     execT     execC    enterT    enterC     exitT     exitC
  0       0.67    122296      0.34    122296      0.16    122296
  1       2.47    360702      1.36    360702      0.54    360702
  2       2.41    369585      1.31    369585      0.53    369585
  3       1.68    246299      0.93    246299      0.37    246299
  *       7.23   1098882      3.94   1098882      1.61   1098882

R00003   CRITICAL        cpp_qsomp2.cpp (233--243)
   001:  [R0001]  cpp_qsomp2.cpp (342--358)
   002:  [R0003]  cpp_qsomp2.cpp (233--243)
 TID     execT     execC    enterT    enterC     exitT     exitC
  0       1.22    255371      0.55    255371      0.31    255371
  1       1.16    242924      0.53    242924      0.30    242924
```

```
2       1.32    278241      0.59    278241      0.34    278241
3       0.98    194745      0.45    194745      0.24    194745
*       4.67    971281      2.13    971281      1.19    971281
```

4 Related Work

A number of performance analysis tools for OpenMP exist. Vendor specific tools such as Intel Thread Profiler [5] and Sun Studio [13] are usually limited to the respective platform but can make use of details of the compiler's OpenMP implementation.

Expert [16] is a tool based on POMP that performs tracing of hybrid MPI and OpenMP applications. After a program run traces are analyzed by Expert which performs an automatic search for patterns of inefficient behavior. Another POMP-based profiler called PompProf is mentioned in [4] but no further details are given.

TAU [7] is also able to profile OpenMP applications by utilizing the Opari instrumenter. TAU additionally profiles user functions, provides support for hardware counters and includes a visualizer for performance results. `ompP` differs in the way performance data is presented. We believe that due to its simplicity, limited purpose and scope, `ompP` might be easier to use for programmers wanting to get an overview of the behavior of their OpenMP codes that the more complex and powerful TAU tool set.

5 Conclusion and Future Work

We have presented our OpenMP profiler `ompP`. The tool can be used to quickly identify regions of inefficient behavior. In fact by analyzing execution counts the tool is also useful for correctness debugging in certain cases (for example to verify that a critical section is actually entered a certain, known number of times for given input data).

An important benefit is the immediate availability of the textual profiling report after the program run, as no further post-processing step is required. Furthermore the tool is naturally very portable and can be used on virtually any platform making it straightforward to compare the performance (and the performance problems) on a number of different platforms.

For the future we are considering the inclusion of hardware performance counters in the data gathering step. Additionally we are investigating to use Tool Gear [9] to be able to related the profiling data to the user's source code in a nice graphical representation.

References

1. Antonio J. Dorta, Casiano Rodríguez, Francisco de Sande, and Arturo Gonzáles-Escribano. The OpenMP source code repository. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2005)*, pages 244–250, February 2005.

2. Michael Gerndt, Bernd Mohr, and Jesper Larsson Träff. Evaluating OpenMP performance analysis tools with the APART test suite. In *Proceedings of the 10th International Euro-Par Conference on Parallel Processing (Euro-Par '04)*, pages 155–162, 2004.
3. Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, 1982.
4. IBM HPC Toolkit `http://www.spscicomp.org/ScicomP10/Presentations/Austin_Klepacki.pdf`.
5. Intel Thread Profiler `http://www.intel.com/software/products/threading/tp/`.
6. Intel Trace Analyzer `http://www.intel.com/software/products/cluster/tanalyzer/`.
7. Allen D. Malony and Sameer Shende. Performance technology for complex parallel and distributed systems. pages 37–46, 2000.
8. Allen D. Malony and Sameer S. Shende. Overhead compensation in performance profiling. In *Proceedings of the 10th International Euro-Par Conference on Parallel Processing (Euro-Par '04)*, pages 119–132, August 2004.
9. John May and John Gyllenhaal. Tool Gear: Infrastructure for parallel tools. In *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '03)*, pages 231–240, 2003.
10. Bernd Mohr, Allen D. Malony, Sameer S. Shende, and Felix Wolf. Towards a performance tool interface for OpenMP: An approach based on directive rewriting. In *Proceedings of the Third Workshop on OpenMP (EWOMP'01)*, September 2001.
11. Bernd Mohr and Jesper Larsson Träff. Initial design of a test suite for automatic performance analysis tools. In *Proc. HIPS*, pages 77–86, 2003.
12. Wolfgang E. Nagel, A. Arnold, M. Weber, Hans-Christian Hoppe, and K. Solchenbach. VAMPIR: visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–90, 1996.
13. Sun Studio `http://developers.sun.com/prodtech/cc/hptc_index.html`.
14. Michael Süß and Claudia Leopold. A user's experience with parallel sorting and openmp. In *Proceedings of the Sixth Workshop on OpenMP (EWOMP'04)*, October 2004.
15. Jeffrey S. Vetter and Frank Mueller. Communication characteristics of large-scal scientific applications for contemporary cluster architectures. *J. Parallel Distrib. Comput.*, 63(9):853–865, 2003.
16. Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. In *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003)*, pages 13–22. IEEE Computer Society, February 2003.