On Using Incremental Profiling for the Performance Analysis of Shared Memory Parallel Applications

Karl Fuerlinger[1], Michael Gerndt[2], and Jack Dongarra[1]

[1] Innovative Computing Laboratory,
Department of Computer Science,
University of Tennessee
{karl, dongarra}@cs.utk.edu
[2] Lehrstuhl für Rechnertechnik und Rechnerorganisation,
Institut für Informatik,
Technische Universität München
gerndt@in.tum.de

Abstract. Profiling is often the method of choice for performance analysis of parallel applications due to its low overhead and easily comprehensible results. However, a disadvantage of profiling is the loss of temporal information that makes it impossible to causally relate performance phenomena to events that happened prior or later during execution. We investigate techniques to add temporal dimension to profiling data by incrementally capturing profiles during the runtime of the application and discuss the insights that can be gained from this type of performance data. The context in which we explore these ideas is an existing profiling tool for OpenMP applications.

1 Introduction

Performance is an important concern for many developers of parallel applications and a large number of tools are available that support can be used for analyzing performance and to identify potential bottlenecks.

Most tools for parallel performance analysis capture data during the execution of an application to later analyze it offline. Depending on the kind of data recorded, *tracing* and *profiling* is commonly distinguished. Profiling denotes the reporting of performance data in a summarized style (such as fraction of total execution time spent in a certain subroutine call) and tracing refers to the recording of individual time-stamped program execution events.

Profiling is often preferred over tracing since it generates less performance data and greatly facilitates a manual interpretation. However, a detailed analysis of the interaction of processes or threads can sometimes require the level of insight only time-stamped event recoding offers. More generally, a significant drawback of profiling is the fact that the *temporal* dimension of performance data is completely lost. That is, with "one-shot" profiles is not possible to relate

performance phenomena in profiling reports to the time when they occurred in the application and to explain the reason and consequences in terms of causal relations with other phenomena or events happening earlier or later.

In this paper we investigate the utility of incremental profiling (or "profiling over time") for performance analysis of parallel applications. We discuss general approaches to add a temporal component to profiling data and the kind of insights that can be derived from incremental profiling. The context in which we explore these ideas is our own profiling tool for OpenMP applications, called `ompP` [1]. Examples that show the utility of our proposed approach come from several applications from the SPEC OpenMP benchmark suite (SPEC OMPM2001) [2].

The rest of this paper is organized as follows: Sect. 2 briefly introduces our profiling tool and describes its existing capabilities. Sect. 3 then lists general options to add temporal dimension to performance data and describes the approach we have taken with `ompP`. Sect. 4 serves as an evaluation of our ideas: we describe the new types of performance data of incremental profiling (often graphical views) that are available to the user and show examples of their utility on application examples that come from the SPEC OpenMP benchmark suite. We describe related work in Sect. 5 and conclude and discuss further directions for our work in Sect. 6.

2 The OpenMP Profiler `ompP`

`ompP` is a profiling tool for OpenMP applications that relies on the Opari instrumenter [3] for source code instrumentation. `ompP` is a static library linked to the target application and delivers a text-based profiling report that is meant to be easily comprehensible by the user at program termination. As opposed to subroutine-based profiling tools like gprof, `ompP` is able to report timings and counts for various OpenMP constructs in terms of the user execution model [4]. As `ompP` is based on instrumentation-added measurement calls (and not on PC sampling, like gprof), it is *exact* in the sense that the reported values are not subject to sampling inaccuracy. An example for the profiling data delivered by `ompP` is shown in Fig. 1.

```
R00002 main.c (34-37) (default) CRITICAL
 TID     execT     execC     bodyT    enterT     exitT   PAPI_TOT_INS
   0      3.00         1      1.00      2.00      0.00           1595
   1      1.00         1      1.00      0.00      0.00           6347
   2      2.00         1      1.00      1.00      0.00           1595
   3      4.00         1      1.00      3.00      0.00           1595
 SUM     10.01         4      4.00      6.00      0.00          11132
```

Fig. 1: An example for `ompP`'s profiling data for an OpenMP `critical` section. The body of this critical section contained a call to `sleep(1.0)` only.

The first line gives the source code location and type of construct. Different counts and timing categories (depending on the particular type of OpenMP construct) are listed as column headers and each line lists the accumulated values for a particular thread, while the last line sums over all threads. `ompP` also supports the measurement of hardware performance counter data for individual threads and OpenMP constructs as well as user-defined regions using PAPI [5]. The user selects a counter to measure via environment variables and the summed counter values then appear as additional columns in the profiles, as shown in Fig 1.

In addition to the flat region profile shown in Fig. 1, `ompP` performs an overhead analysis in which four well-defined overhead classes (synchronization, load imbalance, thread management, limited parallelism) are quantitatively evaluated. Furthermore `ompP` tries to detect common inefficiency situations, such as load imbalance in parallel loops, contention for locks and critical sections, etc. The profiling reports contains a list of the discovered instances of these – so called – *performance properties* [6] sorted by their severity (negative impact on performance).

3 Adding Temporal Dimension to Profiling Data

A straightforward way to add a temporal component to profiling-type performance data is to capture profiles at several points during the execution of the target application (and not just at the end) and to analyze how the profiles change between those capture points. Alternatively (and equivalently), the changes between capture points can be recorded incrementally and the overall state at capture time can later be recovered.

Several trigger events for the collection of profiling reports are conceivable:

- **Timer based, fixed**: Profiles are captured in regular fixed intervals during the execution of the target application. No prior knowledge or modification of the application is required. Since data samples are captured in uniform intervals, they are easy to visualize and comprehend. The overhead with respect to storage space for profiling reports is predicable as it depends on the duration of the program run and the capture interval only.
- **Timer based, adaptive**: This method dynamically adapts the duration between two capture points based on amount of change in profiling data that has been observed. Possible measures for this change are the number of different constructs executed and their invocation count. This option has the potential advantage of offering a finer-grained insight into phases of execution where a lot of change occurs while avoiding profiling reports for phases of largely uniform behavior.
- **Overflow based**: This is another method to correlate the profiling frequency with the activity of the program. With this technique, profiling reports are generated when a hardware counter overflows a pre-set threshold. For floating-point intensive applications, it may for example be beneficial to trigger profiling reports each n floating point operations have occurred.

Other potential triggers for profiling reports are the number of cache misses or the occurrence of page faults.
- **User added**: A method to trigger the generation of profiling reports can be exposed to the user. This technique is especially useful for phase-based programs, where program phases are iteratively executed.

In this paper we investigate the simplest form of incremental profiling described above, capturing profiles in regular, fixed-length intervals during the entire execution time of the application.

4 Evaluation of Incremental Profiling

For both profiling and tracing, the following dimensions of performance data can generally be distinguished:

- Kind of data: Describes which type of data is measured or reported to the user. Examples include timing data, execution counts, performance counter values, and so on.
- Source code location: Data can be collected globally for the whole program or for particular source code entities such as subroutines, OpenMP constructs, basic blocks, individual statements, etc.
- Threads or processes dimension: Measured data can either be reported for individual threads or processes or accumulated over the whole set (by summing or averaging, for example).
- Time dimension: Describes when a particular measurement was made (timestamp) or for which time duration values have been measured.

An appealing property of profiling data is its low dimensionality, i.e., it can often be comprehended textually (like gprof output) or it can be visualized as 1D or 2D graphs in a straightforward way. Adding a new (temporal) dimension jeopardizes this advantage and requires more sophisticated performance data displays. We came up with the following types of useful performance views that can be extracted from the incremental profiling reports delivered by `ompP`:[3]

Performance properties over time: Performance properties [6] offer a very compact way to represent performance analysis knowledge and their change over time can thus be easily visualized. A property like "Imbalance in parallel region `foo.f (23-42)` with severity of 4.5%" carries all relevant context information with it. The severity value denotes the percentage of total execution time improvement that can be expected if the cause for the inefficiency is completely removed. The threads dimension is collapsed in the specification of the property and the source code dimension is encoded as the properties context (`foo.f (23-42)` in the above example).

[3] All examples shown in this section come from an execution of the SPEC OpenMP medium size benchmark suite on a 32 CPU SGI Altix machine, based on Itanium-2 processors with 1.6 GHz and 6 MB L3 cache, used in batch mode.

Properties over time can be visualized as a 2D line-plot, where the x-axis is the time and the y-axis denotes severity values and a line is drawn for each property from the first time it was detected until program termination. Depending on the particular test application, valuable information can be deduced, depending on the behavior of the property graphs. For example, in the example graph shown in Fig. 2 it is evident that the severity of the properties appears to be continuously increasing as time proceeds, indicating that the imbalance situations in this code will become increasingly significant with longer runtime (e.g., larger data sets or more iterations).
Other applications from the SPEC OpenMP benchmark suite showed other interesting features such as initialization routines that generated high initial overheads which amortized over time (i.e., the severity decreased).

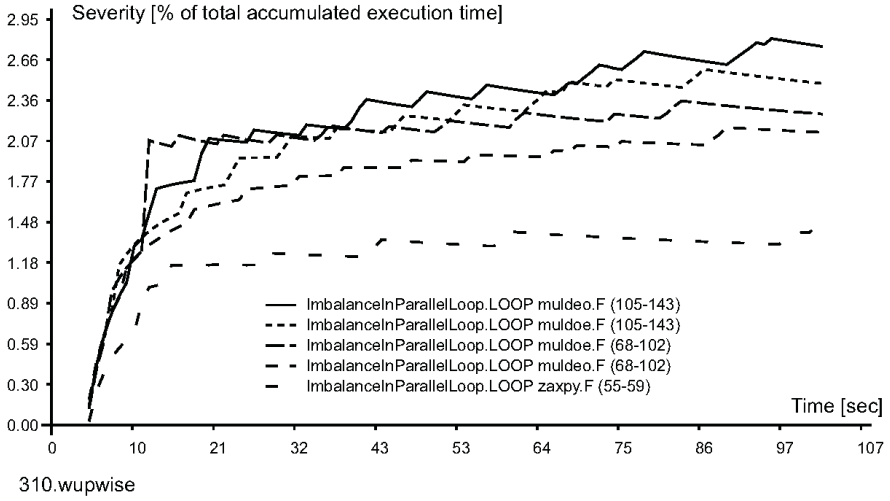


Fig. 2: An example for the "performance properties over time" display for the 310.wupwise application. Shown are the five most severe performance properties.

Region invocations over time: Depending on the size of the test application and the analyst's familiarity with the source code, it can be valuable to know when and how often a particular OpenMP construct, such as a parallel loop, was executed. The region-invocation over time displays offers this functionality.
As shown in Fig. 3 the graph gives the number of invocations of a particular region, this particular case shows the two most time-consuming parallel regions of the 328.fma3d application.
This view is most useful when aggregating (e.g., summing) over all threads, but in certain cases it can be valuable (for critical sections and locks, for example) to see which particular thread executed a construct at which time.

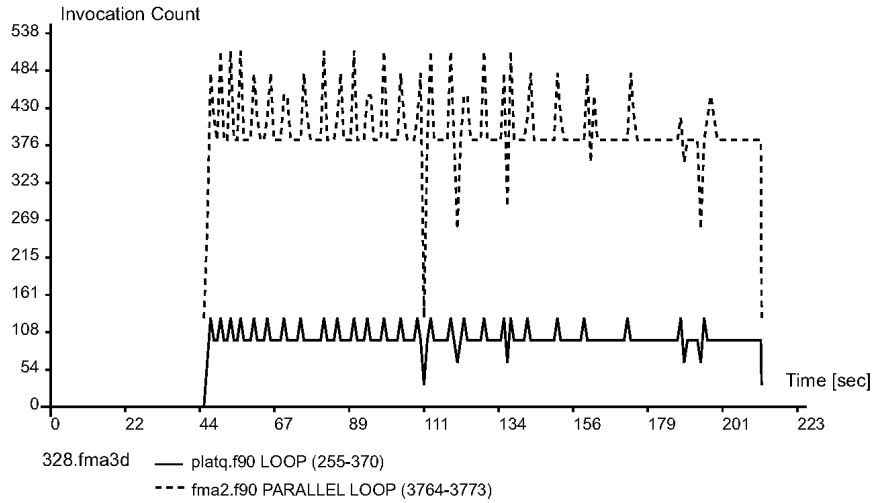


Fig. 3: This graph shows the number of region invocations over time for the 328.fma3d application.

A surface plot can be used for visualization in this case or a heatmap display similar to the one used for visualizing performance counters (see below).

Region execution time over time: This display is similar to the region invocation over time display but shows the execution accumulated execution time between dump intervals instead of the invocation count. Again this display allows the user to see when particular portions of the code actually get executed.

Overheads over time: `ompP` evaluates four overhead classes based on the flat profiling data for individual parallel regions and for the program as a whole. For example, the time required to enter a critical section is attributed to the containing parallel region as synchronization time. A detailed discussion and motivation of this classification scheme can be found in [7].

The overheads over time display plots the incurred overheads (either for a particular parallel region or for the entire application) over the execution timeline of the application, as shown in Fig. 4. This graph gives the overheads as percentage of total aggregated CPU time. Hence, for an execution with 32 CPUs, a overhead percentage of 50 means that 16 CPUs are not doing useful work. The total overhead incurred over the entire program run is thus the integral of the overhead function (area under the graph) and the graph shows when a particular overhead was incurred. In the example in Fig. 4, the most noticeable overhead is synchronization overhead starting at about 30 seconds of execution and lasting for several seconds. A closer examination of the OpenMP profiling reports reveals that this overhead is caused by critical section contention. One thread after the other enters the

critical section and performs a time-consuming initialization operation. This effectively serializes the execution for more than 10 seconds and shows up as a overhead of $31/32 = 97\%$ in the overheads graph.

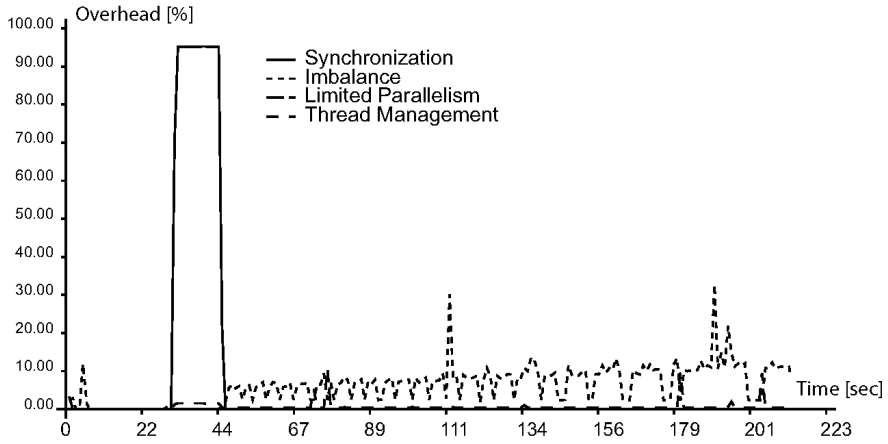
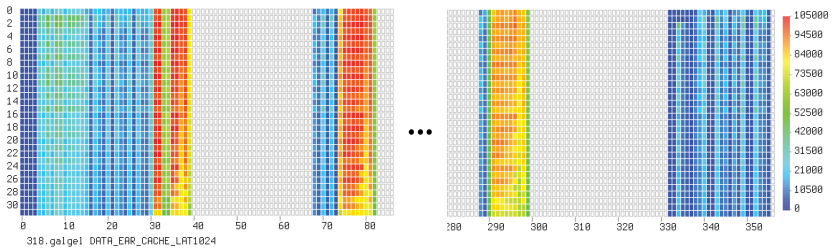


Fig. 4: This graph shows overheads over time for the 328.fma3d application.

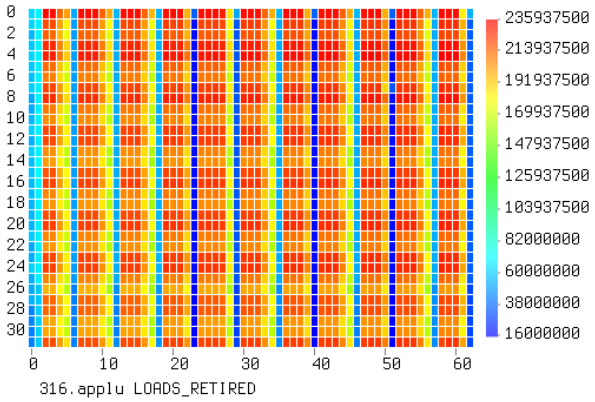
Performance counter heatmaps: This display is used to visualize hardware performance counter values over time and for several threads.

Fig. 5 shows examples of the performance counter heatmap display. The x-axis corresponds to the time (in seconds) while the y-axis corresponds to the thread ID. A color gradient (or gray-scale) coding is indicating high or low counter values. A tile is not filled if no data samples are available for that time period. This type of display is supported for both the entire program as well as for individual OpenMP regions. The example in Fig. 5a shows the `DATA_EAR_CACHE_LAT1024` counter for the SPEC OpenMP application 318.galgel (note that the middle part of the timeline has been cut out to facilitate presentation). This counter measures the number of cache misses that took longer than 1024 cycles to be satisfied and thus roughly corresponds to remote memory accesses on the NUMA architecture of the SGI Alitx machine.
Depending on the selected hardware counters, this view offers very interesting insight into the behavior of the applications. Phenomena that we were able to identify with this kind of performance display include:

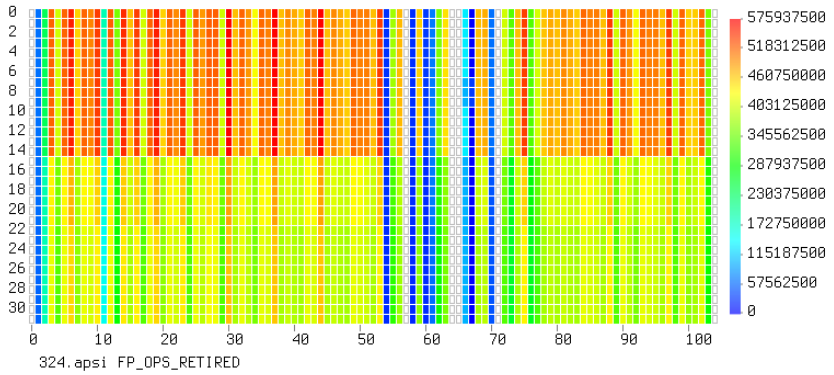
- The homogeneity or heterogeneity of threads. E.g., often threads 16, 8, and 24 would show markedly different behavior compared to other threads in a 32 thread run. Possible reasons for this difference in behavior might be in the application itself (related to the algorithm) but they could also come from the machine organization or system software layer



(a) Cache misses that took longer than 1024 cycles to be satisfied for the 318.galgel application.



(b) Retired load instructions for the 316.applu application.



(c) Retired floating point operations for the 324.apsi application.

Fig. 5: Example performance counter heatmaps. Time is displayed on the horizontal axis (in seconds), the vertical axis lists the threads (32 in this case). The middle part of 5a of the display has been cut out.

(mapping of threads to processors and their arrangement in the machine an its interconnect).
As another example, Fig. 5c gives the number of retired floating point operations for the 324.apsi application and this graph shows a marked difference for processors 0 to 14 vs 15 to 31. We were not able to identify the exact cause for this behavior until now.

- Identification of phase-based behavior, as in Figs. 5a and 5b, some applications show a marked phase-based behavior. It is also evident in many cases that the characteristics of each phase change from iteration to iteration.
- Identification of temporary performance bottlenecks such as short-term bus-contention.

5 Related Work

There are a number of performance analysis tools for OpenMP. Vendor-specific tools such as the Intel Thread Profiler and Sun Studio are usually limited to their respective platform but have the advantage of being able to make use of internal details of the compiler's OpenMP implementation and runtime system. Both the Intel and the Sun tool are based on sampling and can provide the user with some timeline profile displays. Neither of those tools however has a concept similar to `ompP`'s high-level abstraction of performance properties.

TAU [8, 9] is also able to profile and trace OpenMP applications by utilizing the Opari instrumenter. Its performance data visualizer Paraprof supports a number of different profile displays and also supports interactive 3D exploration of performance data, but to the best of our knowledge does not currently have support for a display similar to our performance counter heatmaps. The TAU toolset also contains a utility to convert TAU trace files to profiles which can generate profile series and interval profiles.

OProfile and its predecessor the Digital Continuous Profiling Infrastructure (DCPI) are system-wide statistical profilers based on hardware counter overflows. Both approaches rely on a profiling daemon running in the background and support the dumping of profiling reports at any time. Data acquisition in a style similar to our incremental profiling approach would thus be easy to implement. We are, however, not aware of any study using OProfile or DPCI that investigated continuous profiling for parallel applications. In practice, the necessity of root privileges and the difficulty of relating profiling data back to the user's OpenMP threading model are major stumbling blocks when using those tools. Both issues are not a concern with `ompP` since it is based on source code instrumentation.

6 Conclusion and Future Work

We have presented a study on the utility of incremental profiling for performance analysis of shared memory parallel applications. Our results indicate that valuable information about the temporal behavior of applications can be discovered

by incremental profiling and that this technique strikes a good balance between the level of detail offered by tracing and the simplicity and efficiency of profiling. Using incremental profiling we where able to acquire new insights into the behavior of applications which can due to the lack of temporal data not be gained from pure profiling. The most interesting features are the revelation of iterative behavior, the identification of short-term contention for resources, and the temporal localization of overheads and execution patterns.

Future work is planned in several areas. Firstly, we plan to support other triggers for capturing profiles, most importantly user-added and overflow based. Secondly, be plan to integrate our profiling data with TAU's Paraprof viewer in order to interactively explore the incremental profiling data delivered by `ompP`. Thirdly, we plan to test our ideas in the context of MPI as well, a planned integrated MPI/OpenMP profiling tool based on `mpiP` [10] and `ompP` is the first step in this direction.

References

1. Fuerlinger, K., Gerndt, M.: ompP: A profiling tool for OpenMP. In: Proceedings of the First International Workshop on OpenMP (IWOMP 2005), Eugene, Oregon, USA (May 2005) Accepted for publication.
2. Aslot, V., Eigenmann, R.: Performance characteristics of the SPEC OMP2001 benchmarks. SIGARCH Comput. Archit. News **29**(5) (2001) 31–40
3. Mohr, B., Malony, A.D., Shende, S.S., Wolf, F.: 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)
4. Itzkowitz, M., Mazurov, O., Copty, N., Lin, Y.: An OpenMP runtime API for profiling Accepted by the OpenMP ARB as an official ARB White Paper available online at `http://www.compunity.org/futures/omp-api.html`.
5. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.J.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. **14**(3) (2000) 189–204
6. Gerndt, M., Fürlinger, K.: Specification and detection of performance problems with ASL. Concurrency and Computation: Practice and Experience (2006) To appear.
7. Fuerlinger, K., Gerndt, M.: Analyzing overheads and scalability characteristics of OpenMP applications. In: Proceedings of the Seventh International Meeting on High Performance Computing for Computational Science (VECPAR'06), Rio de Janeiro, Brasil (2006) To appear.
8. Shende, S.S., Malony, A.D.: The TAU parallel performance system. International Journal of High Performance Computing Applications, ACTS Collection Special Issue (2005)
9. Malony, A.D., Shende, S.S., Bell, R., Li, K., Li, L., Trebon, N.: Advances in the TAU performance analysis system. In Getov, V., Gerndt, M., Hoisie, A., Malony, A., Miller, B., eds.: Performance Analysis and Grid Computing, Kluwer (2003) 129–144
10. Vetter, J.S., Mueller, F.: Communication characteristics of large-scale scientific applications for contemporary cluster architectures. J. Parallel Distrib. Comput. **63**(9) (2003) 853–865