ELSEVIER

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2010) 1–8

Procedia Computer Science

International Conference on Computational Science, ICCS 2010

OpenMP Application Profiling – State of the Art and Directions for the Future

Karl Fürlinger

Computer Science Division, EECS Department, University of California at Berkeley, Soda Hall 515, Berekely, U.S.A.

Abstract

OpenMP is a successful approach to writing threaded parallel applications. This article describes the state of the art in performance profiling OpenMP applications, covering vendor performance tools and platform independent techniques. The features of the OpenMP profiler ompP are described in detail and an outlook of future directions in this area is given.

Keywords: OpenMP application profiling, Overhead analysis, Scalability analysis

1. Introduction

OpenMP is a successful approach for writing shared-memory thread-parallel programs. It currently enjoys a renewed interest in high performance computing and other areas due to the shift towards multicore computing and it is fully supported by every major compiler. OpenMP is also being promoted as one of the recommended high-level programming approaches for GPU accelerator computing such as for Intel's upcoming Larrabee GPU architecture [1].

As with any programming model, developers are interested in improving the efficiency of their codes and a number of performance tools are available to help in understanding the execution characteristics of applications. However, the lack of a standardized monitoring interface often renders the performance analysis process more involved, complex, and error prone than it is for other programming models.

OpenMP is a language extension (and not a library, like MPI) and the compiler is responsible for transforming the program into threaded code. As a consequence, even a detailed analysis of the execution of the threads in the underlying low level threading package is of limited use to understand the behavior on the OpenMP level. For example, compilers frequently transform parallel regions into outlined routines and the mapping of routine names (the machine's model of the execution) to the original source code (the user's model of the execution) is nontrivial. Vendor tools (paired with a compiler) can overcome this problem because they are aware of the transformations a compiler can employ and the naming scheme used for generating functions. Portable tools can't have this kind of intricate knowledge for every compiler and OpenMP runtime and have to resort to other mechanisms, discussed later in this article.

The rest of this paper is organized as follows: Sect. 2 discusses the vendor-provided solutions for OpenMP profiling, while Sect. 3 describes how platform independent monitoring can be achieved using a source-to-source translator

Email address: `fuerling@eecs.berkeley.edu` (Karl Fürlinger)

called Opari. Sect. 4 introduces the profiling tool ompP which is built on Opari to gather a variety of useful execution metrics for OpenMP applications. Sections 5 and 6 discuss the analysis of execution overheads and the scalability of applications based on the data delivered by ompP, respectively. Sect. 7 concludes with an outlook on the path forward.

2. Profiling Tools supplied by Vendors

OpenMP is today supported by the compilers of every major computer vendor and the GNU compiler collection has OpenMP support since version 4.2. Most vendors also offer performance tools, such as the Intel Thread Profiler [2], Sun Studio [3], or Cray Pat [4], supporting users in analyzing the performance of their OpenMP codes.

The vendor tools mostly rely on statistical sampling and take advantage of the knowledge of the inner workings of the runtime and the transformations applied by the compiler. Typical performance issues that can be successfully detected by these tools are situations of insufficient parallelism, locking overheads, load imbalances, and in some cases memory performance issues such as false sharing.

A proposal for a platform independent standard interface for profiling was created by SUN [5]. With this proposal, tools could register for notifications from the runtime and ask for it to supply the mapping from the runtime to user execution model. The proposal is the basis of SUN's own tool suite and is mostly used for statistical profiling there. The proposal is not part of the OpenMP specification but was accepted as an official white paper by the OpenMP Architecture Review Board (ARB). So far acceptance of the proposal outside of SUN has been limited, however, and independent tool developers have to resort to other methods.

3. Platform Independent Monitoring

All approaches for monitoring OpenMP code in a platform independent way in tools such as ompP [6], Vampir [7], [8], TAU [9], [10], KOJAK [11], and Scalasca [12] today use source code instrumentation. This approach has some shortcomings, for example it requires the user to recompile their code for instrumentation. However it has been found in practice to work well for many applications and the usage of automated preprocessor scripts eases the burden for developers.

The monitoring approach used by all the aforementioned tools is based on a source code preprocessor called Opari [13]. Opari is a part of the KOJAK and Scalasca toolsets and it adds calls inside and around OpenMP pragmas according to the POMP specification [14]. An example source code fragment with Opari instrumentation is shown in Fig. 1: In this example the user code is a simple parallel region (lines 2, 5, 6, and 12 in bold font represent the original user code), all other lines are added by Opari. `POMP_Parallel_{fork|join}` calls are placed on the outside of the parallel construct and `POMP_Parallel_{begin|end}` calls are placed as the first and last statements inside the parallel region, respectively. An additional explicit barrier and corresponding `POMP_Barrier_{enter|exit}` calls (lines 8–10) are placed towards the end of the parallel region as well. Doing so converts the implicit barrier at the end of the parallel region into an explicit barrier that can subsequently be monitored by tools, for example to detect load imbalances among threads.

4. OpenMP Profiling with ompP

ompP [6] is a profiling tool for OpenMP based on source-code instrumentation using Opari. ompP implements the POMP interface to monitor the execution of the instrumented application. An application is linked with ompP's profiling library and on program termination a profiling report is written to a file. Environment variables can be used to influence various settings of the data collection and reporting process.

Consider a simple OpenMP program fragment containing only a critical section inside a parallel region and a call to `sleep(1.0)` in the critical section. Fig. 2 shows the corresponding profile for the critical section: we see that this critical section was unnamed (for named critical sections the name would appear instead of "default") and that it is located at lines 34-37 in a file named "main.c". ompP's internal designation for this region is R00002.

The profile represents an execution with four threads. Each line displays data for a particular thread and the last line sums over all threads. A number of columns depict the actual measured profiling values. Column names that end in a capital "C" represent counts while columns ending with a capital "T" represent times. Evidently each thread

```
POMP_Parallel_fork [master]              ] enter          ] main
#pragma omp parallel {                   |                |
    POMP_Parallel_begin [team]           ]      ] body    |
                                                |         |
        /* user code in                         |         |
           parallel region */                   |         |
                                                |         |
        POMP_Barrier_enter [team]        ] barr ]         |
        #pragma omp barrier              |                |
        POMP_Barrier_exit [team]         ]                |
    POMP_Parallel_end [team]             ] exit           |
}                                        |                |
POMP_Parallel_join [master]              ]                ]
```

Figure 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 subregion nesting used by `ompP`.

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

Figure 2: An example of `ompP`'s profiling data for an OpenMP `critical` section. The body of this critical section contains only a call to `sleep(1.0)`.

executed the critical construct and that the total execution time (`execT`) varies from 1.0 second to 4.0 seconds. `bodyT` is the time inside the body of the critical construct and it is 1.0 seconds corresponding to the time in `sleep()`, while `enterT` is the time (if any) threads have to wait before they can enter the critical section. In the example in Fig. 2 thread 1 was the first one to enter, then thread 2 entered after waiting 1.0 seconds, then thread 0 after waiting 2.0 seconds, and so on.

`ompP` reports timings and counts in the style shown in Fig. 2 for all OpenMP constructs. Table 1 shows the full table of constructs supported by `ompP` and the timing and count categories used in the profiling report. `execT` and `execC` are always reported and represent the entire construct. An additional breakdown is provided for some constructs, such as the `startupT` and `shutdwnT` for the time required starting up and tearing down the threads of a parallel region, respectively. This breakdown is based on the subdivision of regions into subregions as shown on the right hand side of Fig. 1. If applicable, a region is thought of being composed of a subregion for entering (`enter`), exiting (`exit`), a main body (`body`), and a barrier (`barr`), and `main` = `enter` + `body` + `barr` + `exit` always holds.

In addition to the basic flat region profiles such as the ones shown in Fig. 2, `ompP` supports a number of more productivity features.

- `ompP` records the application's call graph and the nesting of the OpenMP regions and computes inclusive and exclusive times from it. This allows performance data for the same region involved from different execution paths to be analyzed separately.

- It allows the monitoring of hardware performance counter values using PAPI [15], [16]. If a PAPI installation is available on a machine, `ompP` will include the necessary functionality to set up, start, and stop the counters for all OpenMP constructs. Hardware counters can be used with `ompP` by setting the environment variable `OMPP_CTR`*n* to the name of PAPI predefined or platform-specific event names. For example:

	main		*enter*		*body*					*barr*	*exit*	
construct	`execT`	`execC`	`enterT`	`startupT`	`bodyT`	`sectionT`	`sectionC`	`singleT`	`singleC`	`exitBarT`	`exitT`	`shutdwnT`
`MASTER`	•	•										
`ATOMIC`	• S	•										
`BARRIER`	• S	•										
`FLUSH`	• S	•										
`USER REGION`	•	•										
`CRITICAL`	•	•	• S		•						• M	
`LOCK`	•	•	• S		•						• M	
`LOOP`	•	•			•					• I		
`WORKSHARE`	•	•			•					• I		
`SECTIONS`	•	•				•	•			• I/L		
`SINGLE`	•	•						•	•	• I		
`PARALLEL`	•	•		• M	•					• I		• M
`PARALLEL LOOP`	•	•		• M	•					• I		• M
`PARALLEL SECTIONS`	•	•		• M		•	•			• I/L		• M
`PARALLEL WORKSHARE`	•	•		• M	•					• I		• M

Table 1: Timing and count categories reported by `ompP` for various OpenMP constructs and overhead classification of the times reported by `ompP` into one of the four overhead classes. Synchronization (S), Load Imbalance (I), Thread Management (M), Limited Parallelism (L).

```
$> export OMPP_CTR1=PAPI_L2_DCM
```

requests the monitoring of level 2 data cache misses.

- `ompP` computes the parallel coverage (i.e., the fraction of total execution time spent in parallel regions). According to Amdahl's law [17] a high parallel coverage is quintessential to achieve satisfactory speedups and high parallel efficiency as the number of threads increases.
- Another feature of `ompP` is continuous and incremental profiling [18]. Instead of capturing only a single profile at the end of the execution, this technique allows an attribution of profile features on the timeline axis without the overheads typically associated with tracing.
- A further recent technique aimed at providing temporal runtime information without storing full traces is capturing and analyzing the execution control flow of applications. The method described in [19] shows that this can be achieved with low overhead and small modifications to the data collection process. The resulting flow graphs are designed for interactive exploration together with performance data in the form of timings and hardware performance counter statistics to determine phase-bases and iterative behavior [20].

5. Overheads Analysis

Any parallel algorithm and the execution of a parallel program will usually involve some overheads limiting the scalability and the parallel efficiency. Typical sources of overheads are synchronization (the need to serialize the access to a shared resource, for example) or any time spent in setting up and destroying the threads of parallel execution.

An example of this is also shown in Fig. 2: `enterT` is the time threads have to wait to enter the critical section. From an application's standpoint threads idle and don't do useful work on behalf of the application and hence `enterT` constitutes a form of synchronization overhead (since it arises due to the fact that the threads have to synchronize their activity). Considering the timings reported by `ompP`, a total of four different overhead classes can be identified.

Synchronization: Overheads that arise because threads need to coordinate their activity. An example is the waiting time to enter a critical section or to acquire a lock.

Imbalance: Overhead due to different amounts of work performed by threads and subsequent idle waiting time, for example in work-sharing regions.

Limited Parallelism: This category represents overhead resulting from unparallelized or only partly parallelized regions of code. An example is the idle waiting time threads experience while one thread executes a `single` construct.

Thread Management: Time spent by the runtime system for managing the application's threads. That is, time for creation and destruction of threads in parallel regions and overhead incurred in critical sections and locks for signaling the lock or critical section as available.

Table 1 shows the classification of the timings reported by `ompP` into no overhead (●) or one of the four overhead classes: S for synchronization overhead, I for imbalance, L for Limited Parallelism, and T for Thread management overhead.

`ompP`'s application profile includes an overhead analysis report which offers various interesting insights into where an application wastefully spends its time. The overhead analysis report contains an overview of the parallel regions in a program and a breakdown into work and the four overhead classes for each parallel region separately and for the program as a whole. The overhead breakdown forms the basis for the scalability analysis.

6. Scalability Analysis

The scalability of an application or of individual parallel regions within an application can be analyzed in a detail by computing an overhead breakdown while varying the number of OpenMP threads for the execution.

Assuming a constant workload and increasing number of threads (i.e., a strong scaling study), one can plot scalability graphs like the one shown in Fig. 3. Here the number of threads is plotted on the horizontal axis and the total aggregated execution time (summed over all threads) is plotted on the vertical axis. The topmost line is derived from the overall wallclock execution time of an application. From this total time one can subtract the four overhead classes (synchronization, load imbalance, thread management, and limited parallelism) for each thread count. A perfectly scaling code (linear decrease in the execution time the number of threads is increased) would result in a horizontal line in this type of scalability graph. A line inclining from the horizontal corresponds to imperfect scaling and a declining from the horizontal points towards superlinear speedup.

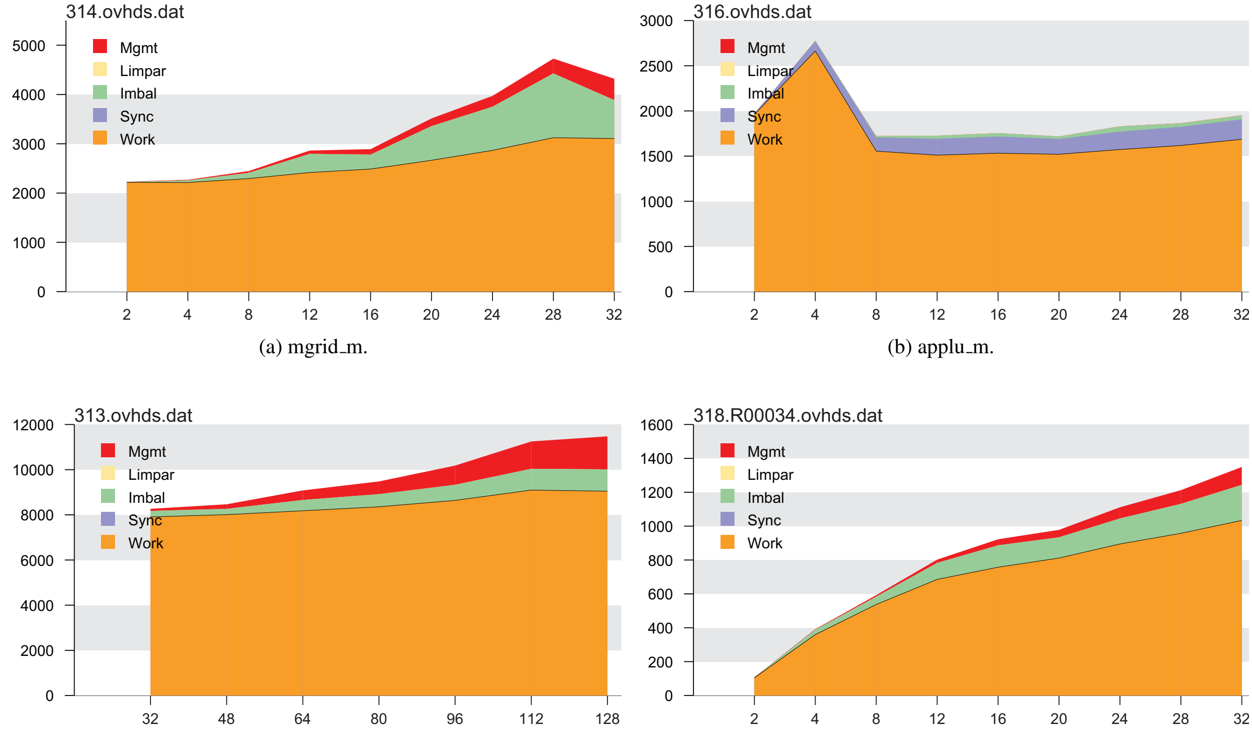
Figures 3a, 3b, and 3c are scalability graphs of full applications from the SPEC OpenMP benchmark suite, corresponding to the "mgrid", "applu" and "swim" applications, respectively. The experiments presented in Fig. 3 were run on SGI Alitx ccNUMA machines. The scaling studies from 2 to 32 threads were done on a 32 processor SGI Alitx 3700 Bx2 machine (1.6 GHz, 6 MByte L3-Cache) while the 32–128 thread study has been performed on a larger Altix 4700 machine with the same type of processor.

Evidently the scaling behavior of these applications is widely different:

mgrid: This code scales relatively poorly (cf. Fig. 3a). Almost all of the application's 12 parallel loops contribute to the bad scaling behavior with increasingly severe load imbalance. As shown in Fig. 3a, there appears to be markedly reduced load imbalance for 32 and 16 threads. Investigating this issue further we discovered that this behavior is only present in three of the application's parallel loops (`mgrid.f 265-301`, `mgrid.f 317-344`, and `mgrid.f 360-384`). A source-code analysis of these loops reveals that in all three instances, the loops are always executed with an iteration count that is a power of two (which ranges from 2 to 256 for the `ref` dataset). Hence, thread counts that are not powers of two generally exhibit more imbalance than powers of two.

applu: The interesting scalability graph of this application shows super-linear speedup. This behavior can be attributed exclusively to one parallel region (`ssor.f 138-209`) in which most of the execution time is spent (this region contributes more than 80% of total execution time), the other parallel regions do not show a super-linear speedup. To investigate the reason for the super-linear speedup we used `ompP`'s ability to measure hardware performance counters. By common wisdom, the most likely cause of super-linear speedup is the increase in overall cache size that allows the application's working set to fit into the cache for a certain number of processors. To test this hypothesis we measured the number of L3 cache misses incurred in the `ssor.f 138-209` region and the results indicate that, in fact, this is the case. The total number of L3 cache misses (summed over all threads) is at 15 billion for 2 threads, and at 14.8 billion at 4 threads. At 8 threads the cache misses reduce to 3.7 billion, at 12 threads they are at 2.0 billion from where on the number stays approximately constant up to 32 threads.

swim: The dominating source of inefficiency in this application is thread management overhead that dramatically increases in severity from 32 to 128 threads. Its main source is the reduction of three scalar variables in the small parallel loop `swim.f 116-126`. At 128 threads more than 6 percent of total accumulated application runtime are spent in this reduction operation. The time for the reduction is actually larger than the time spent in the body of this parallel loop.



(a) mgrid_m.

(b) applu_m.

(c) swim_l.

(d) galgel_m (lapack.f90 5081-5092).

Figure 3: Scalability graphs for some of the applications of the SPEC OpenMP benchmark suite. Suffix _m refers to the medium size benchmark while suffix _l denotes large size benchmark. The horizontal axis denotes processor (thread) count and the vertical is the accumulated execution time (over all threads) in seconds.

In addition to scalability graphs for the whole application, ompP offers the option to analyze the scaling of individual parallel regions. As the workload characteristics of parallel regions within a program may vary considerably, so can their scalability characteristics and each region might have a different operating sweet spot in terms of computing throughput or energy efficiency.

Fig. 3d shows an example for a scalability graph of an individual region of the application "galgel". Evidently this region scales very badly. In terms of the wallclock execution time, the 32 thread version execution of this region is only 22% faster than the 2 thread execution and the parallel efficiency at 32 threads is only 8% – almost no benefit is gained from using more than 2 threads for this region.

A more detailed and complete discussion of the scaling properties of several of the SPEC OpenMP benchmarks can be found in [21].

7. Future Directions

OpenMP is a continually evolving standard and new features pose new requirements and challenges for performance profiling. A recent major feature introduced with version 3.0 of the OpenMP specification is *tasking* – the ability to dynamically specify and enqueue small chucks of work for later execution.

A preliminary study [22] investigated the feasibility of extending the source code based instrumentation approach to handle the new tasking related constructs. The resulting modified version of `ompP` could successfully handle many common use cases for tasking. The notion of overheads had to be adapted, however, to reflect the fact that with tasking threads reaching the end of a parallel region are no longer idle but can in fact start fetching and executing tasks from the task pool.

One use case, that of *untied* tasks, proved to be a challenge. With untied tasks, an executing thread can stop and resume task execution at any time and in fact a different thread can pick up where the previous thread left off. These events are impossible to be observed purely based on source code instrumentation without a standardized callback mechanism that would notify a tool of such an occurrence.

To overcome these issues in particular and to simplify usage of `ompP` in general, tying together instrumentation sources from multiple layers, such as source code instrumentation, interposing on the native threading library and incorporating the SUN whitepaper seems like a promising way for the path forward.

References

[1] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan, Larrabee: A many-core x86 architecture for visual computing, ACM Trans. Graph. 27 (3) (2008) 1–15.

[2] Intel Thread Profiler `http://www.intel.com/software/products/threading/tp/`.

[3] Sun Studio `http://developers.sun.com/prodtech/cc/hptc_index.html`.

[4] Using Cray performance analysis tools. `http://docs.cray.com/books/S-2376-41/S-2376-41.pdf`.

[5] M. Itzkowitz, O. Mazurov, N. Copty, Y. Lin, 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`.

[6] K. Fürlinger, M. Gerndt, ompP: A profiling tool for OpenMP, in: Proceedings of the First International Workshop on OpenMP (IWOMP 2005), Eugene, Oregon, USA, 2005.

[7] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, K. Solchenbach, VAMPIR: Visualization and analysis of MPI resources, Supercomputer 12 (1) (1996) 69–90.

[8] H. Brunst, B. Mohr, Performance analysis of large-scale OpenMP and hybrid MPI/OpenMP applications with VampirNG, in: Proceedings of the First International Workshop on OpenMP (IWOMP 2005), Eugene, Oregon, USA, 2005.

[9] A. D. Malony, S. S. Shende, Performance technology for complex parallel and distributed systems (2000) 37–46.

[10] S. S. Shende, A. D. Malony, The TAU parallel performance system, International Journal of High Performance Computing Applications, ACTS Collection Special Issue.

[11] F. Wolf, B. 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), IEEE Computer Society, 2003, pp. 13–22.

[12] M. Geimer, F. Wolf, B. J. N. Wylie, B. Mohr, Scalable parallel trace-based performance analysis, in: Proceedings of the 13th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI 2006), Bonn, Germany, 2006, pp. 303–312.

[13] B. Mohr, A. D. Malony, S. S. Shende, F. Wolf, Towards a performance tool interface for OpenMP: An approach based on directive rewriting, in: Proceedings of the Third Workshop on OpenMP (EWOMP'01), 2001.

[14] B. Mohr, A. D. Malony, H.-C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger, S. Shah, A performance monitoring interface for OpenMP, in: Proceedings of the Fourth Workshop on OpenMP (EWOMP 2002), Rome, Italy, 2002.

[15] PAPI web page: `http://icl.cs.utk.edu/papi/`.

[16] S. Browne, J. Dongarra, N. Garner, G. Ho, P. J. Mucci, A portable programming interface for performance evaluation on modern processors, Int. J. High Perform. Comput. Appl. 14 (3) (2000) 189–204.

[17] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Readings in computer architecture, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000, pp. 79–81, reprint of a work originally published in 1967.

[18] K. Fürlinger, S. Moore, Continuous runtime profiling of OpenMP applications, in: Proceedings of the 2007 Conference on Parallel Computing (PARCO 2007), 2007, pp. 677–686.

[19] K. Fürlinger, S. Moore, Visualizing the program execution control flow of OpenMP applications, in: Proceedings of the 4th International Workshop on OpenMP (IWOMP 2008), Purdue, Indiana, USA, 2008, pp. 181–190, lNCS 5004.
URL `http://www.cs.utk.edu/~karl/research/pubs/FUERLINGER_2008_CFG.pdf`

[20] K. Fürlinger, S. Moore, Detection and analysis of iterative behavior in parallel applications, in: Proceedings of the 2008 International Conference on Computational Science (ICCS 2008), Krakow, Poland, 2008, pp. 261–267, lNCS 5103.
URL `http://www.cs.utk.edu/~karl/research/pubs/FUERLINGER_2008_Phases.pdf`

[21] K. Fürlinger, M. Gerndt, J. Dongarra, Scalability analysis of the SPEC OpenMP benchmarks on large-scale shared memory multiprocessors, in: Proceedings of the 2007 International Conference on Computational Science (ICCS 2007), Beijing, China, 2007, pp. 815–822.

[22] K. Fürlinger, D. Skinner, Performance profiling for OpenMP tasks, in: Proceedings of the 5th International Workshop on OpenMP (IWOMP 2009), Dresden, Germany, 2009.