Note: This is a preprint of a publication to appear in the conference proceedings of the VECPAR'06 conference, pending selection. VECPAR'06 was held in Rio de Janeiro, July 10–13, 2006.

Analyzing Overheads and Scalability Characteristics of OpenMP Applications*

Karl Fürlinger and Michael Gerndt

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

Abstract. Analyzing the scalability behavior and the overheads of OpenMP applications is an important step in the development process of scientific software. Unfortunately, few tools are available that allow an exact quantification of OpenMP related overheads and scalability characteristics. We present a methodology in which we define four overhead categories that we can quantify exactly and describe a tool that implements this methodology. We evaluate our tool on the OpenMP version of the NAS parallel benchmarks.

1 Introduction

OpenMP has emerged as the standard for shared-memory parallel programming. While OpenMP allows for a relatively simple and straightforward approach to parallelizing an application, it is usually less simple to ensure efficient execution on large processor counts.

With the widespread adoption of multi-core CPU designs, however, scalability is likely to become increasingly important in the future. The availability of 2-core CPUs effectively doubles the number of processor cores found in commodity SMP systems based for example on the AMD Opteron or Intel Xeon processors. This trend is likely to continue, as the road-maps of all major CPU manufacturers already include mulit-core CPU designs. In essence, performance improvement is increasingly going to be based on parallelism instead of improvements in single-core performance in the future [13].

Analyzing and understanding the scalability behavior of applications is therefore an important step in the development process of scientific software. Inefficiencies that are not significant at low processor counts may play a larger role when more processors are used and may limit the application's scalability. While it is straightforward to study how execution times scale with increasing processor numbers, it is more difficult to identify the possible reasons for imperfect scaling.

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

Here we present a methodology and a tool to evaluate the runtime characteristics of OpenMP applications and to analyze the overheads that limit scalability at the level of individual parallel regions and for the whole program. We apply our methodology to determine the scalability characteristics of several benchmark applications.

2 Methodology

To analyze the scalability of OpenMP applications we have extended our OpenMP profiler `ompP` [6] with overhead classification capability. `ompP` is a profiler for OpenMP programs based on the POMP interface [9] that relies on source code instrumentation by Opari [10]. `ompP` determines execution counts and times for all OpenMP constructs (parallel regions, work-sharing regions, critical sections, locks, ...) in the target application. Depending on the type of the region different timing and count categories are reported.

`ompP` consists of a monitoring library that is linked to an OpenMP application. Upon termination of the target application, `ompP` writes a profiling report to a file. An example output of `ompP` for a critical section region is shown in Fig. 1. A table that lists the timing categories reported by `ompP` for the different region types is shown in Fig. 2, a particular timing is reported if a "•" is present, the counts reported by `ompP` are not shown in Fig. 2.

```
R00002   CRITICAL        cpp_qsomp1.cpp (156-177)
 TID     execT     execC    enterT     bodyT     exitT
  0       1.61    251780      0.87      0.43      0.31
  1       2.79    404056      1.54      0.71      0.54
  2       2.57    388107      1.38      0.68      0.51
  3       2.56    362630      1.39      0.68      0.49
  *       9.53   1406573      5.17      2.52      1.84
```

Fig. 1: Example `ompP` output for an OpenMP `CRITICAL` region. `R00002` is the region identifier, `cpp_qsomp1.cpp` is the source code file and `156-177` denotes the extent of the construct in the file. Execution times and counts are reported for each thread individually, and summed over all threads in the last line.

The timing categories reported by `ompP` shown in Fig. 2 have the following meaning:

- `seqT` is the sequential execution time for a construct, i.e., the time between forking and joining threads for `PARALLEL` regions and for combined work-sharing parallel regions as seen by the master thread. For a `MASTER` region it similarly represents the execution time of the master thread only (the other threads do not execute the `MASTER` construct).

	seqT	execT	bodyT	exitBarT	enterT	exitT
MASTER	•					
ATOMIC		• (S)				
BARRIER		• (S)				
USER_REGION		•				
LOOP		•		• (I)		
CRITICAL		•	•		• (S)	• (M)
LOCK		•	•		• (S)	• (M)
SECTIONS		•	•	• (I/L)		
SINGLE		•	•	• (L)		
PARALLEL	•	•		• (I)	• (M)	• (M)
PARALLEL_LOOP	•	•		• (I)	• (M)	• (M)
PARALLEL_SECTIONS	•	•	•	• (I/L)	• (M)	• (M)

Fig. 2: The timing categories reported by ompP for the different OpenMP constructs and their categorization as overheads by ompP's overhead analysis. (S) corresponds to synchronization overhead, (I) represents overhead due to imbalance, (L) denotes limited parallelism overhead, and (M) signals thread management overhead.

- execT gives the total execution time for constructs that are executed by all threads. The time for thread n is available as execT $[n]$. execT always contains bodyT, exitBarT, enterT and exitT.
- bodyT is the time spent in the "body" of the construct. This time is reported as singleBodyT for SINGLE regions and as sectionT for SECTIONS regions.
- exitBarT is the time spent in "implicit exit barriers". I.e., in worksharing and parallel regions OpenMP assumes an implicit barrier at the end of the construct, unless a nowait clause is present. Opari adds an explicit barrier to measure the time in the implicit barrier.
- enterT and exitT are the times for entering and exiting critical sections and locks. For parallel regions enterT is reported as startupT and corresponds to the time required to spawn threads. Similarly, exitT is reported as shutdownT and represents thread teardown overhead.

2.1 Overhead Analysis

From the per-region timing data reported by ompP we are able to analyze the overhead for each parallel region separately, and for the program as a whole. We have defined four overhead categories that can be exactly quantified with the profiling data provided by ompP:

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 that results 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 (see below for a more detailed discussion).

The table in Fig. 2 details how timings are attributed to synchronization (S), imbalance (I), limited parallelism (L), thread management overhead (M), and work (i.e., no overhead). This attribution is motivated as follows:

- `exitBarT` in work-sharing or parallel regions is considered imbalance overhead, except for `single` regions, where the reason for the time spent in the exit barrier is assumed to be limited parallelism. The time in the exit barrier of a `sections` construct is either imbalance or limited parallelism, depending on the number of `section` constructs inside the `sections` construct, compared to the number of threads. If there are fewer sections than threads available, the waiting time is considered limited parallelism overhead and load imbalance otherwise.
- The time spent in `barrier` and `atomic` constructs is treated as synchronization overhead.
- The time spent waiting to enter a critical section or to acquire a lock is considered synchronization overhead. Opari also adds instrumentation to measure the time spent for leaving a critical section and releasing a lock. These times reflect the overhead of the OpenMP runtime system to signal the lock or critical section being available to waiting threads. Hence, these overheads do not relate to the synchronization requirement of the threads but rather represent an overhead related to the implementation of the runtime system. Consequently, the resulting waiting times are treated as thread management overhead.
- The same considerations as above hold true for `startupT` and `shutdownT` reported for parallel regions. This is the overhead for thread creation and destruction, which is usually insignificant, except in cases where a team of threads is created and destroyed repeatedly (if, for example, a small `parallel` region is placed inside a loop). Again, this overhead is captured by in the thread management category.

The overheads for each category are accumulated for each parallel region in the program separately. That is, if a parallel region P contains a critical section C, C's enter time will appear as synchronization overhead in P's overhead statistics. Note that, while `ompP` reports inclusive timing data in its profiling reports, the timing categories related to overheads are never nested and never overlap. Hence, a summation of each sub-region's individual overhead time gives the correct total overhead for each parallel region.

An example of ompP's overhead analysis report is shown in Fig. 3 (the columns corresponding to the thread management overhead category are omitted due to space limitations). The first part of the report, denoted by ⓐ, gives general information about the program run. It lists the total number of parallel regions (OpenMP parallel constructs and combined parallel work-sharing constructs), the total wallclock runtime and the parallel coverage (or parallel fraction). The parallel coverage is defined as the fraction of wallclock execution time spent inside parallel regions. This parameter is useful for estimating the optimal execution time according to Amdahl's law on p processors as

$$T_p = \frac{T_1 \alpha_1}{p} + T_1(1 - \alpha_1),$$

where T_i is the execution time on i processors and α_1 is the parallel coverage of an execution with one thread.

Section ⓑ lists all parallel regions of the program with their region identifiers and location in the source code files, sorted by their wallclock execution time.

Part ⓒ shows the parallel regions in the same order as in part ⓑ and details the identified overheads for each category as well as the total overhead (`Ovhds` column). The total runtime is given here accumulated over all threads (i.e., `Total` = wallclock runtime × number of threads) and the percentages for the overhead times shown in parenthesis refer to this runtime.

The final part in the overhead analysis report (ⓓ) lists the same overhead times but the percentages are computed according to the total runtime of the program. The regions are also sorted with respect to their overhead in this section. Hence, the first lines in section ⓓ show the regions that cause the most significant overall overhead as well as the type of the overhead. In the example shown in Fig. 3, the most severe inefficiency is imbalance overhead in region `R00035` (a parallel loop in `y_solve.f, lines 27-292`) with a severity of 3.44%.

2.2 Scalability Analysis

The overhead analysis report of ompP gives valuable insight into the behavior of the application. From analyzing overhead reports for increasing processor counts, the scalability behavior of individual parallel regions and the whole program can be inferred. We have implemented the scalability analysis as a set of scripts that take several ompP profiling reports as input and generate data to visualize the program's scalability as presented in Sect. 3.

The graphs show the change of the distribution of overall time for increasing processor counts. That is, the total execution time as well as each overhead category is summed over all threads and the resulting accumulated times are plotted for increasing processor numbers.

3 Evaluation

To evaluate the usability of the scalability analysis as outlined in this paper, we test the approach on the OpenMP version of the NAS parallel benchmarks [8]

```
---------------------------------------------------------------
-----     ompP Overhead Analysis Report     ----------------------
---------------------------------------------------------------

Total runtime (wallclock)   : 736.82 sec [4 threads]
Number of parallel regions  : 14
Parallel coverage           : 736.70 sec (99.98%)

Parallel regions sorted by wallclock time:
         Type                              Location      Wallclock (%)
R00018  parall                      rhs.f (16-430)     312.48 (42.41)
R00037   ploop                  z_solve.f (31-326)     140.00 (19.00)
R00035   ploop                  y_solve.f (27-292)      88.68 (12.04)
R00033   ploop                  x_solve.f (27-296)      77.03 (10.45)
...
     *       *                                   *     736.70 (99.98)

Overheads wrt. each individual parallel region:
         Total      Ovhds (%) =  Synch  (%)  + Imbal   (%)  + Limpar (%)
R00018 1249.91   0.44 ( 0.04)  0.00 ( 0.00)   0.35 ( 0.03)  0.00 ( 0.00)
R00037  560.00 100.81 (18.00)  0.00 ( 0.00) 100.72 (17.99)  0.00 ( 0.00)
R00035  354.73 101.33 (28.56)  0.00 ( 0.00) 101.24 (28.54)  0.00 ( 0.00)
R00033  308.12  94.62 (30.71)  0.00 ( 0.00)  94.53 (30.68)  0.00 ( 0.00)
...

Overheads wrt. whole program:
         Total      Ovhds (%) =  Synch  (%) +  Imbal   (%) +  Limpar (%)
R00035  354.73 101.33 ( 3.44)  0.00 ( 0.00) 101.24 ( 3.44)  0.00 ( 0.00)
R00037  560.00 100.81 ( 3.42)  0.00 ( 0.00) 100.72 ( 3.42)  0.00 ( 0.00)
R00033  308.12  94.62 ( 3.21)  0.00 ( 0.00)  94.53 ( 3.21)  0.00 ( 0.00)
...
     * 2946.79 308.52 (10.47)  0.00 ( 0.00) 307.78 (10.44)  0.00 ( 0.00)
```

Fig. 3: Example overhead analysis report generated by `ompP`, the columns related to the thread management category (`Mgmt`) are omitted due to space limitations.

(version 3.2, class "C"). Most programs in the NAS benchmark suite are derived from CFD applications, it consists of five kernels (EP, MG, CG, FT, IS) and three simulated CFD applications (LU, BT, SP). Fig. 4 shows the characteristics of the benchmark applications with respect to the OpenMP constructs used for parallelization.

	BT	CG	EP	FT	IS	LU	MG	SP
MASTER	4					13	2	4
ATOMIC	2		1			2		1
BARRIER					1	3		
LOOP	25	13	1		1	30	5	25
CRITICAL								1
LOCK								
SECTIONS								
SINGLE						6		
PARALLEL	6	9	1		2	8	2	6
PARALLEL_LOOP	4	5	1	8	2	1	8	8
PARALLEL_SECTIONS								

Fig. 4: The OpenMP constructs found in the NAS parallel benchmarks version 3.2.

Fig. 6 presents the result of the scalability analysis performed 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. The number of threads was increased from 2 to 32. The graphs in Fig. 6 show the accumulated runtime over all threads. Hence, a horizontal line corresponds to a perfectly scaling code with ideal speedup. For convenience, a more familiar *speedup graph* (with respect to the 2-processor run) computed from the same data is shown in Fig. 5.

In Fig. 6, the total runtime is divided into work and the four overhead categories, and the following conclusions can be derived:

- Overall, the most significant overhead visible in the NAS benchmarks is imbalance, only two applications show significant synchronization overhead, namely IS and LU.
- Some applications show a surprisingly large amount of overhead, as much as 20 percent of the total accumulated runtime is wasted due to imbalance overhead in SP.
- Limited parallelism does not play a significant role in the NAS benchmarks. While not visibly discernable in the graphs in Fig. 6 at all, the overhead is actually present for some parallel regions albeit with very low severity.
- Thread management overhead is present in the applications, mainly in CG, IS and MG, although it is generally of little importance.
- EP scales perfectly, it has almost no noticeable overhead.
- The "work" category increases for most applications and does not stay constant, even though the actual amount of work performed is independent of

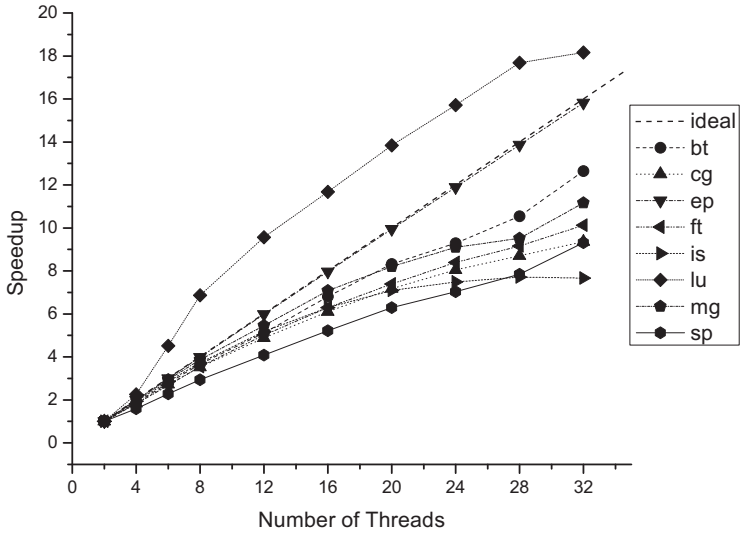
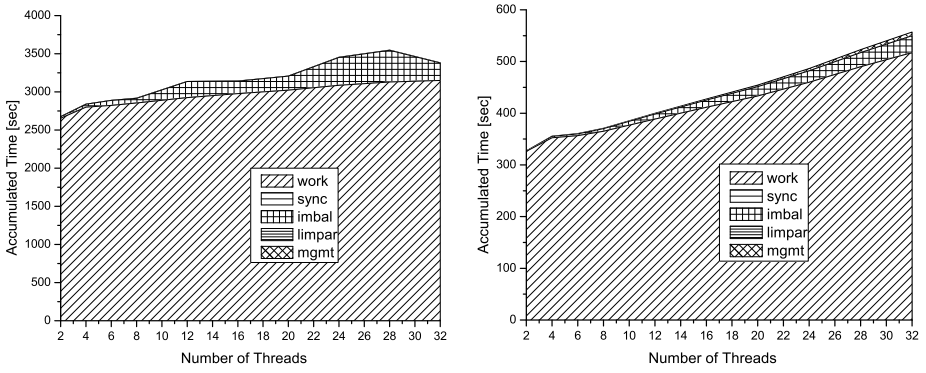


Fig. 5: Speedup achieved by the NAS benchmark programs relative to the 2-processor execution.

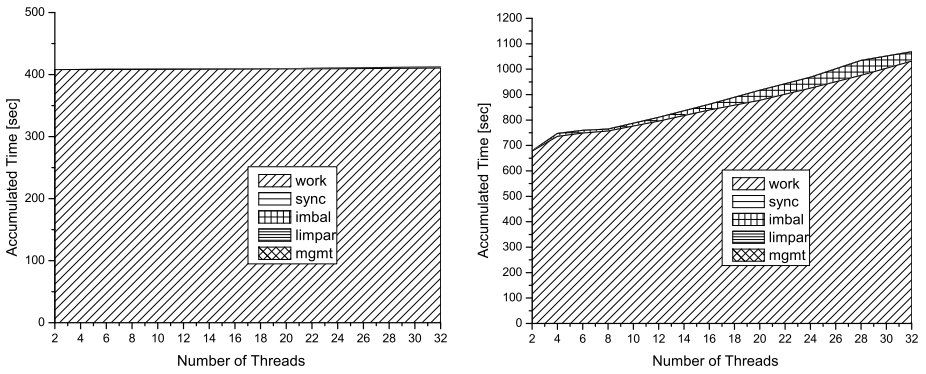
the number of threads used. This can be explained with overhead categories that are currently not accounted for by `ompP`, for example increasing memory access times at larger processor configurations. Additional factors that influence the work category are the increasing overall cache size (which can lead to super-linear speedups) and an increased rate of cache conflicts at larger systems. This issue is discussed further in Sect. 5.

- For some applications, the summed runtime increases linearly, for others it increases faster than linearly (e.g., IS scales very poorly).
- For LU, the performance increases super-linearly at first, then at six processors the performance starts to deteriorate. The reason for the super-linear speedup is most likely the increased overall cache size.

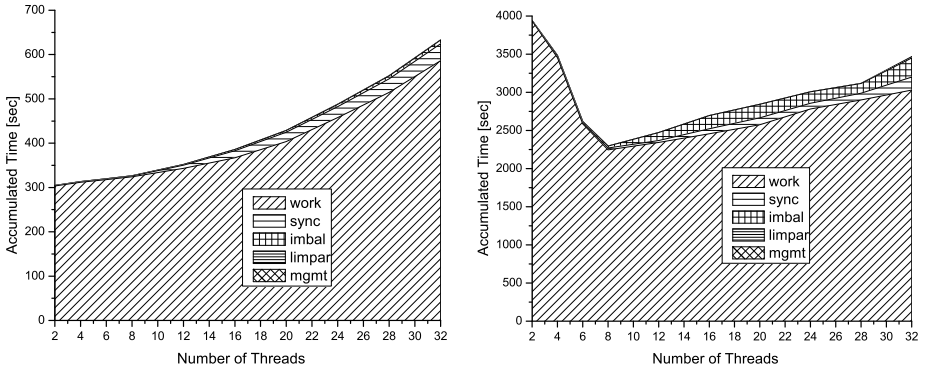
`ompP` also allows for a scalability analysis of individual parallel regions. An example for a detailed analysis of the BT benchmark is shown in Fig. 7. The left part shows the scalability of the work category (without overheads), while the right part shows the total overhead (all categories summed) for the four most time consuming parallel regions. It is apparent that for powers of two, the overhead (which is mainly imbalance) is significantly less than it is for other configurations. A more detailed analysis of `ompP`'s profiling report and the application's source code reveals the cause: most overhead is incurred in loops with an iteration count of 160, which is evenly divisible by 2, 4, 8, 16, and 32.



(a) BT. (b) CG.

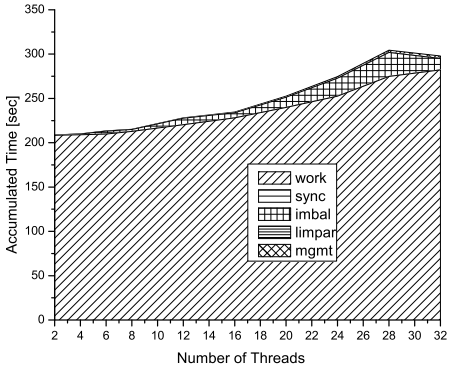


(c) EP. (d) FT.

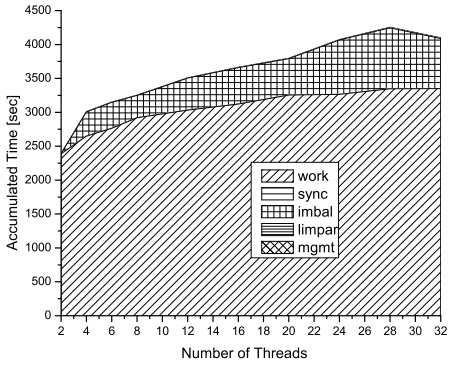


(e) IS. (f) LU.

Fig. 6: Scaling of total runtime and the separation into work and overhead categories for the NAS OpenMP parallel benchmarks (BT, CG, EP, FT, IS, and LU).

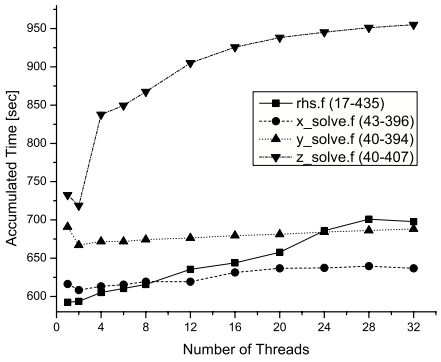


(g) MG.

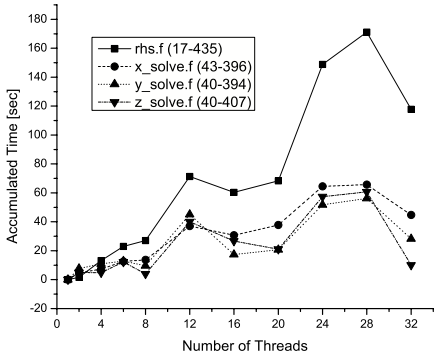


(h) SP.

Fig. 6: Scaling of total runtime and the separation into work and overhead categories for the NAS OpenMP parallel benchmarks (MG and SP).



(a) Work performed in the four most important parallel regions of BT.



(b) Total overhead incurred in the four most important parallel regions of BT.

Fig. 7: Detailed scalability analysis at the level of individual parallel regions of the BT application. The four most important parallel regions are analyzed with respect to the work performed and the overheads incurred for each region individually.

4 Related Work

Mark Bull describes a hierarchical classification scheme for temporal and spatial overheads in parallel programs in [3]. The scheme is general (not dependant on a particular programming model) and strives to classify overheads in categories that are complete, meaningful, and orthogonal. Overhead is defined as the difference between the observed performance on p processors and the "best possible" performance on p processors. Since the best possible performance is unknown (it can at best be estimated by simulation), $T_p^{ideal} = \frac{T_1}{p}$ is often used as an approximation for the ideal performance on p processors. Thus

$$T_p = T_p^{ideal} + \sum_i O_p^i \tag{1}$$

where O_p^i represent the overhead in category i. This is similar to our scheme with the difference that `ompP` does not report overheads with respect to the wallclock execution time but aggregated over all threads.

Bull's hierarchical classification scheme has four categories at the top level:

- Information Movement
- Critical Path
- Control of Parallelism
- Additional Computation

While this scheme allows for a well defined conceptual breakdown of where an application spends its time, `ompP`'s classification scheme is based on what can be actually automatically be measured. For example, it is not possible to account for additional computation automatically.

Bane and Riley developed a tool called Ovaltine [1, 2] that performs overhead analysis for OpenMP code. The overhead scheme of Ovaltine is based on the classification scheme by Bull. Ovaltine performs code instrumentation based on the Polaris compiler. Not all overheads in Ovaltine's scheme can be computed automatically. For example the cost of acquiring a lock has to be determined empirically. Practically, only the "load imbalance" and "unparallelized" overheads are computed automatically in Ovaltine as described in [1].

Scal-Tool [11] is a tool for quantifying the scalability bottlenecks of shared memory codes. The covered bottlenecks include insufficient cache, load imbalance and synchronization. Scal-Tool is based on an empirical model using cycles-per-instruction (CPI) breakdown equations. From a number of measurement (fixed data-set, varying number of processors and varying the size of the dataset on a single processor, the parameters in the CPI equations can be estimated. The result of the analysis is a set of graphs that augment the observed scalability graph of the application with estimated scalability, if one or more of the scalability bottlenecks (cache, load imbalance, synchronization) are removed.

Scalea [14] is a tool for performance analysis of Fortran OpenMP, MPI and HPF codes. Scalea computes metrics based on a classification scheme derived from Bull's work. Scalea is able to perform overhead-to-region analysis and

region-to-overhead analysis. I.e., show a particular overhead category for all regions or show all overhead categories for a specific region.

Fredrickson et al. [5] have evaluated the performance characteristics of the class B of the NAS OpenMP benchmarks version 3.0 on a 72 processor Sun Fire 15K. The speedup of the NAS benchmarks is determined for up to 70 threads. In their evaluation, CG shows super-linear speedup, LU shows perfect scalability, FT scales very poorly and BT SP and MG show good performance (EP and IS are not evaluated). In contrast, in our study CG shows relatively poor speedup while LU shows super-linear speedup. Our results for FT, BT, SP, and MG are more or less in-line with theirs.

Fredrickson et al. also evaluate "OpenMP overhead" by counting the number of parallel regions and multiplying this number with an empirically determined overhead for creating a parallel region derived from an execution of the EPCC micro-benchmarks [4]. The OpenMP overhead is low for most programs, ranging from less than one percent to five percent of the total execution time, for CG the estimated overhead is 12%. Compared to our approach this methodology of estimating the OpenMP overhead is less flexible and accurate, as for example it does not account for load-imbalance situations and requires an empirical study to determine the "cost of a parallel region". Note that in `ompP` all OpenMP-related overheads are accounted for, i.e., the work category does not contain any OpenMP related overhead.

Finally, vendor-specific tools such as Intel Thread Profiler [7] and Sun Studio [12] often implement overhead classification schemes similar to `ompP`. However, these tools are limited to a particular platform, while `ompP` is compiler and platform-independent and can thus be used for cross-platform overhead comparisons, for example.

5 Summary and Future Work

We presented a methodology for overheads- and scalability analysis of OpenMP applications that we integrated in our OpenMP profiler `ompP`. We have defined four overhead categories (synchronization, load imbalance, limited parallelism and thread management) that are well defined and can explicitly be measured. The overheads are reported per parallel region and for the whole program. `ompP` allows for an exact quantification of all OpenMP related overheads.

From the overhead reports for increasing processor counts we can see how programs scale and how the overheads increase in importance. We have tested the approach on the NAS parallel benchmarks and were ably to identify some key scalability characteristics.

Future work remains to be done to cover further overhead categories. What is labeled Work in Fig. 5 actually contains overheads that are currently unaccounted for. Most notably it would be important to account for overheads related to memory access. Issues like accessing a remote processors memory on a ccNUMA architecture like the SGI Altix and coherence cache misses impact performance negatively while increased overall caches size helps performance and

can actually lead to negative overhead. For the quantification of these factors we plan to include support for hardware performance counters in `ompP`.

References

1. Michael K. Bane and Graham Riley. Automatic overheads profiler for OpenMP codes. In *Proceedings of the Second Workshop on OpenMP (EWOMP 2000)*, September 2000.
2. Michael K. Bane and Graham Riley. Extended overhead analysis for OpenMP (research note). In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing (Euro-Par '02)*, pages 162–166. Springer-Verlag, 2002.
3. J. Mark Bull. A hierarchical classification of overheads in parallel programs. In *Proceedings of the First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 208–219, London, UK, 1996. Chapman & Hall, Ltd.
4. J. Mark Bull and Darragh O'Neill. A microbenchmark suite for OpenMP 2.0. In *Proceedings of the Third Workshop on OpenMP (EWOMP'01)*, September 2001.
5. Nathan R. Fredrickson, Ahmad Afsahi, and Ying Qian. Performance characteristics of OpenMP constructs, and application benchmarks on a large symmetric multiprocessor. In *Proceedings of the 17th ACM International Conference on Supercomputing (ICS 2003)*, pages 140–149. ACM Press, 2003.
6. Karl Fürlinger and Michael Gerndt. ompP: A profiling tool for OpenMP. In *Proceedings of the First International Workshop on OpenMP (IWOMP 2005)*, May 2005. Accepted for publication.
7. Intel Thread Profiler `http://www.intel.com/software/products/threading/tp/`.
8. H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, 1999.
9. Bernd Mohr, Allen D. Malony, Hans-Christian Hoppe, Frank Schlimbach, Grant Haab, Jay Hoeflinger, and Sanjiv Shah. A performance monitoring interface for OpenMP. In *Proceedings of the Fourth Workshop on OpenMP (EWOMP 2002)*, September 2002.
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. Yan Solihin, Vinh Lam, and Josep Torrellas. Scal-Tool: Pinpointing and quantifying scalability bottlenecks in DSM multiprocessors. In *Proceedings of the 1999 Conference on Supercomputing (SC 1999)*, Portland, Oregon, USA, November 1999.
12. Sun Studio `http://developers.sun.com/prodtech/cc/hptc_index.html`.
13. Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.
14. Hong-Linh Truong and Thomas Fahringer. SCALEA: A performance analysis tool for parallel programs. *Concurrency and Computation: Practice and Experience*, (15):1001–1025, 2003.