A Quantitative Analysis of OpenMP Task Runtime Systems

Sascha Hunold[0000−0002−5280−3855] and Klaus Kraßnitzer[0000−0003−1217−1029]⋆

Research Group for Parallel Computing
Faculty of Informatics, TU Wien
Vienna, Austria
{hunold,krassnitzer}@par.tuwien.ac.at

Abstract. Although OpenMP is heavily used to parallelize for-loops, it also supports task-parallel programming, which is important for parallelizing irregular applications. In this work, we focus on the performance of OpenMP runtime systems for task-based applications. In particular, we investigate the performance of different OpenMP runtime systems when scheduling a large set independent tasks of different granularity. To that end, we propose a new OpenMP benchmark, which features profiling and tracing options that help developers to reason about the observed performance differences. We compare the execution times measured for a variety of compilers, such as gcc, icc, clang, aocc, and pgcc, for both homogeneous and heterogeneous workloads. Our study shows that there are significant performance differences between the different OpenMP implementations. We also show that the performance attainable with a compiler strongly depends on the machine architecture, the number of threads, the thread-pinning strategy, and the task granularity.

Keywords: OpenMP tasks · benchmarking · scheduling.

1 Introduction

In high-performance computing (HPC), OpenMP is the de-facto standard for parallelizing applications at the level of a compute node. In this work, we focus on parallel OpenMP applications that run on shared-memory parallel, multi-core machines. The most common type of today's multi-core machines are cache-coherent NUMA systems (ccNUMA), i.e., the multi-core processors typically have several DRAM memory controllers and partitioned last-level caches (e.g., Level 3 data caches). As a consequence of this ccNUMA architecture, the latency for reading from and writing to memory depends on the actual location of a core and the memory address. The traditional way of parallelizing applications with OpenMP is marking the compute-heavy for-loops with specific OpenMP pragmas. Compilers are then able to transform the programs into data-parallel fork-join applications, where each thread is responsible for specific chunks of

⋆ This work was partially supported by the Austrian Science Fund (FWF): project P 33884-N.

the overall, global iteration space. The other, later introduced parallelization strategy in OpenMP is task-parallel programming. This strategy is particularly helpful for parallelizing recursive computational patterns or irregular applications in general, where tasks of different type and granularity can be dispatched by individual threads. The concept of OpenMP tasks increases the potential degree of parallelism that can be exploited by programmers, yet they also increase the scheduling complexity for the OpenMP runtime systems.

In this paper, we want to answer the question of how efficiently current C/C++ compilers (with OpenMP support) handle a large number of parallel tasks. To that end, we propose the OMPTB benchmark suite, which contains three basic OpenMP task processing strategies (inspired by the EPCC microbenchmarks [1]):

1. `MasterTask`: the master thread creates all tasks, but tasks are executed by all worker threads,
2. `ParallelTask`: all threads both emit and execute tasks, and
3. `ParallelFor`: the actual task code is executed using a single parallel for-loop. No OpenMP tasks are created, but the same number of instructions is executed. This strategy serves as a performance baseline.

We perform a large set of experiments with these three task processing strategies and address the following questions:

- How efficiently do current compilers process a large number of tasks? The compilers examined in this study are: gcc, icc, aocc, clang, and pgcc.
- How much does the task granularity (i.e., the runtime of each task) impact the performance difference between the different compilers?
- How do the different compilers deal with heterogeneous tasks, i.e., if the runtime of the tasks varies significantly?
- What is the impact of the thread mapping strategy on the performance of the task benchmark?
- How is the task workload balanced across the threads? Our hypothesis is that the more balanced the tasks are across the different threads the shorter the runtime should be (cf. Terboven et al. [13]).
- Is the runtime of the task benchmark correlated with the number of cache misses? This is a reasonable assumption considering the fact that we run on highly partitioned ccNUMA systems where compute nodes have up to 16 NUMA nodes.

In this paper, we make the following contributions:

- We present a benchmark for assessing the performance of OpenMP runtime systems. The benchmark features profiling and tracing capabilities, which help to investigate performance differences between compilers. The design of the benchmark is compiler-fair, i.e., the code executed by each OpenMP task is compiled with one fixed compiler to avoid assembly differences, while the OpenMP part is compiled with every investigated OpenMP compiler.
- We present an in-depth experimental study of the performance difference of various compilers for scheduling OpenMP tasks on shared-memory systems.

– We assess the performance of the OpenMP runtime systems when scheduling OpenMP tasks in the presence of heterogeneous tasks.

The remainder of the paper is structured as follows. In Sect. 2, we discuss the related work and indicate how previous work has influenced our benchmarking setup. In Sect. 3, we give a brief overview of our benchmark suite OMPTB, before we explain our experimental setup in Sect. 4. As we put an emphasis on empirical results, we show a large set of experiments in Sect. 5 and draw conclusions in Sect. 6.

2 Related Work

Several other works have already evaluated OpenMP task runtime systems in different contexts. A pioneering work in this field was published by Bull et al. [1], who proposed a set of OpenMP benchmarks to evaluate the cost of applying OpenMP pragmas in various settings. In particular, they proposed the `taskbench` benchmark, which can be used to assess the overhead associated with creating and processing OpenMP tasks. Bull et al. [1] use the sequential time for executing a loop with a fixed work W as the reference time. They measure the time to execute the same loop with various task creation strategies, e.g., only the master threads or all threads create a set of tasks. For evaluating the overhead of using OpenMP tasks, `taskbench` performs weak-scaling experiments, i.e., the number of tasks per thread that is created initially stays constant. The time difference between the parallel execution of work pW on p cores and the sequential execution of work W is called the overhead. In contrast, in our present work, we focus on a strong scaling analysis and keep the overall work exactly the same in each experiment. The benchmark `taskbench` fixes the "work time" of a loop iteration, i.e., how long each iteration should take. In order to estimate the waiting time, Bull et al. [1] use a nested busy loop that executes k iterations, where these k iterations should match this waiting/work time. In `taskbench`, this value of k is estimated every time the benchmark starts, leading to variances of the so-created homogeneous workload between different experimental runs.

Terboven et al. [13] compared how well different OpenMP implementations execute task-parallel OpenMP codes on NUMA machines. They compared performance results obtained with compilers from Intel, GNU, Oracle, and PGI. Similar to our approach, they investigated how load imbalance impacts the performance. To this end, they created a specific heterogeneous workload, where the time for executing a task increases linearly with the number of tasks. Each task internally reads data from memory to examine both load imbalance and data locality effects on NUMA machines. They showed that spawning OpenMP tasks concurrently by all threads often leads to a better performance than if only one thread is creating the tasks.

Olivier et al. [10] analyzed the scalability behavior of different task scheduling systems. In particular, they compared the performance results obtained with Intel's icc and GNU's gcc to the ones obtained with the Qthreads library. The Qthreads library allows them to use different scheduling strategies at different

levels of the NUMA architecture, i.e., they have an implementation with a single LIFO queue or with multiple queues and different work stealing strategies. They showed strong scaling results for a variety of benchmarks from the Barcelona OpenMP Tasks Suite (BOTS) [4].

Clet-Ortega et al. [3] presented an orthogonal work to Olivier et al. [10], where the authors analyze different scheduling strategies of OpenMP tasks on NUMA systems in their own customizable OpenMP runtime system called MPC. The idea is that the number of task queues could be a parameter, e.g., there could be one queue per system, one queue per socket, or one queue per core. The authors studied the performance of the different granularity options for the tasks queues and different work stealing strategies for BOTS applications.

Schuchart et al. [12] extended the EPCC OpenMP MicroBenchmark Suite to analyze the performance of OpenMP task runtimes in the presence of task dependencies. To that end, they defined several task dependency patterns, which were evaluated independently.

Gautier et al. [6] investigated the internal overheads of managing tasks in OpenMP. They instrumented the LLVM OpenMP runtime libOMP to measure the delay of different steps in the task creation process. The authors examined the impact of internal implementation choices on the performance, such as the maximum length of task queues or the hashtable size. Gautier et al. [6] also reported that a substantial fraction of the overhead can be attributed to checking task dependencies (in case dependent tasks are used).

Several commonly used multicore benchmarks are collected in the PARSEC benchmark suite [15]. Since task-based programming has gained importance, task-centric modifications of the PARSEC benchmarks were devised to examine the scaling behavior when expressing the parallel work in the form of tasks [2, 8].

Lastly, Yang and He [14] present an extensive survey on work stealing approaches in the context of task-parallel programming on multicore machines.

3 OMPTB: The OpenMP Task Benchmark

Now, we introduce the OpenMP Task Benchmark (OMPTB) and the supported task creation strategies.[1] The overall design and structure of the micro-benchmarks have been inspired by the works of Bull et al. [1], Terboven et al. [13], and Olivier et al. [10].

Micro-benchmark Structure Our main objective is to create a stress test for the OpenMP scheduling system. Therefore, we focus on scheduling a large number of independent tasks onto a set of homogeneous cores. In Graham's scheduling notation, we are interested in the problems $\mathrm{P}\,||\,C_{\max}$ and $\mathrm{P}\,|\,\bar{p}_i{=}\bar{p}\,|\,C_{\max}$ [7], where $\bar{p}_i = \bar{p}$ is a special case, in which all tasks (jobs) have the same running time $\bar{p}$. From an implementation standpoint, we would like to create the simplest way of testing the scheduling system with the smallest amount of noise introduced

[1] `https://github.com/parlab-tuwien/omp-task-bench`

Listing 3.1: Version `MasterTask`

```
#pragma omp parallel firstprivate(m)
{
#pragma omp master
  for (i = 0; i < n; i++) {
    if( hetero_workload )
      m = get_work(i);
#pragma omp task firstprivate(m)
    res[ridx] = add_bench(m);
  }
#pragma omp taskwait
}
```

Listing 3.2: Version `ParallelTask`

```
#pragma omp parallel firstprivate(m)
{
#pragma omp for
  for (i = 0; i < n; i++) {
    if( hetero_workload )
      m = get_work(i);
#pragma omp task firstprivate(m)
    res[ridx] = add_bench(m);
  }
#pragma omp taskwait
}
```

Listing 3.3: Version `ParallelFor`

```
#pragma omp parallel firstprivate(m)
{
#pragma omp for
  for (i = 0; i < n; i++) {
    if( hetero_workload )
      m = get_work(i);
    res[ridx] = add_bench(m);
  }
}
```

by experimental factors. In our context, a scheduling instance for homogeneous tasks is defined by three variables: n denotes the number of tasks to be created, m denotes the work done in each task, and p denotes the number of threads to be created. Since each thread is mapped to one core exclusively, p also defines the number of cores used to schedule this instance. We also consider the more general case, where each task can have a different amount of work. In this heterogeneous case, the work of each task is drawn randomly from a given distribution, which will be discussed later (cf. Sect. 4).

In order to test the scheduling system, our benchmark executes n jobs of work m on p cores. Our benchmark suite supports two commonly used task creation strategies:

- `MasterTask`: The master thread creates all n tasks, as shown in Listing 3.1.
- `ParallelTask`: All p threads create all n tasks, which is outlined in Listing 3.2.

As a performance baseline, we execute the same n function calls, instead of using OpenMP tasks, in a parallel For-loop (cf. `ParallelFor` in Listing 3.3). Since all loop iterations are independent, the parallel For-loop will provide a lower bound on the performance of the task scheduling system in the case of homogeneous tasks.

Workload Options The actual work of each task is done by the `add_bench` function, which takes m as an input and computes $\sum_i^m i$ in a for loop, and the result is returned as a double value. When we test the heterogeneous problem instances, the value of m is selected independently for each task. The `add_bench` function should mimic a real function call, and thus, it returns an actual result. In our

Table 1: Multi-core machines and compilers used in our study

machine	*Hydra*	*Nebula*	*VSC-5*
processor	Intel Xeon 6130F	AMD EPYC 7551	AMD Epyc 7713
core frequency	2.10 GHz	2.00 GHz	2.00 GHz
nb of sockets	2	2	2
nb of cores per node	32	64	128
compilers	gcc 12.1.0	gcc 12.1.0	gcc 11.2.0
	clang 14.0.4	clang 14.0.4	clang 12.0.1
	pgcc 22.5	pgcc 22.5	pgcc 22.5
	icc 2021.7.0	aocc 3.2.0	aocc 3.2.0

benchmark, we always store the result of the `add_bench` function in a variable, which the compiler cannot optimize away. However, if we just stored the latest result of each `add_bench` function from all threads in one global variable, we would create a false sharing issue among the threads. Therefore, each thread stores the latest result of `add_bench` in its own part of the global `res` array. The index `ridx` ensures that each thread accesses a different cache line.

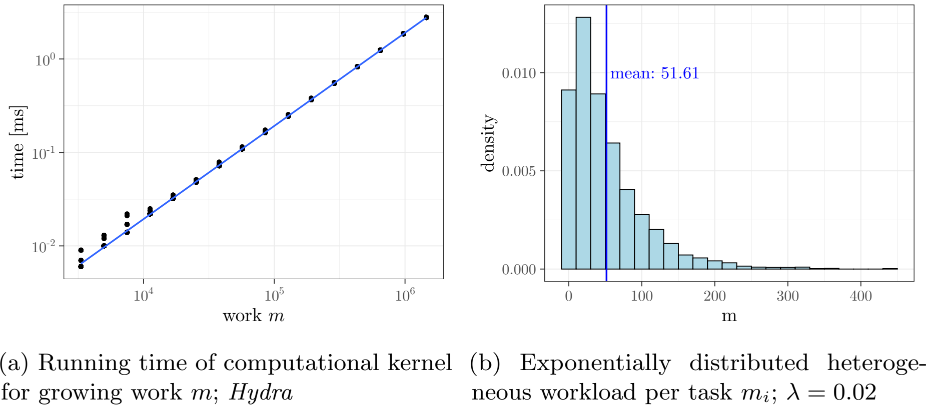
Due to its simplicity, our benchmark setup has two advantages compared to the previous benchmarks. First, the work m is always the same for different executions of the benchmark, which reduces noise and improves reproducibility. Second, the `add_bench` function only needs to read one integer value (m) from memory, and thus, the benchmark is insensitive to the different bandwidths that typically occur between the various NUMA nodes on current multi-core processors.

Considerations for Compiler Fairness Our study should reveal differences in the OpenMP runtime systems when scheduling a large number of independent tasks. Therefore, the actual code that each task executes should be exactly the same. For this reason, we compile the task's code into a separate, dynamic library with exactly one compiler (gcc in all cases), to ensure that the assembly code of the individual tasks is identical. The rest of the benchmark, in particular the OpenMP pragmas, is compiled with each tested compiler.

4 Experimental Setup

Now, we explain our experimental, hardware, and software setup for comparing various OpenMP runtime systems.

Hardware and Software Setup We conduct experiments on three different multi-core, shared-memory systems that comprise 32, 64, and 128 physical cores, which are called *Hydra*, *Nebula*, and *VSC-5*, respectively. We provide an overview of the hardware and software details in Table 1. On the Intel system, *Hydra*, we compare the Intel icc compiler to the compilers gcc, pgcc, and clang, while on the AMD systems, *Nebula* and *VSC-5*, we use the aocc compiler instead of the Intel icc.



(a) Running time of computational kernel for growing work m; *Hydra*

(b) Exponentially distributed heterogeneous workload per task m_i; $\lambda = 0.02$

Fig. 1: Example workloads for parameter m

Since aocc is built on top of the clang infrastructure and also uses `libomp`, we expect similar performance results from clang and aocc. The most important difference between the Intel processor and both AMD processors, which are used in our experiments, is the number of NUMA nodes. The Intel processor only has one NUMA node per socket, while the AMD processors have either four (*Nebula*) or two (*VSC-5*) NUMA nodes per socket.

Workload Options A central parameter of our benchmark is the work done per task. If the work is small, the fraction of the overall time spent in the OpenMP runtime system grows, and differences in the scheduling methods become more pronounced. Figure 1a shows how the runtime of our `add_bench` function depends on the work parameter m. In this experiment, we increase the work m and measure the running time of `add_bench` for each m. We can observe that the runtime grows linearly with m, exactly as it should.

As already mentioned, we also investigate how well the OpenMP runtime systems perform for heterogeneous workloads. In particular, we would like to answer whether the scheduling results change if the workload is heterogeneous. Feitelson [5] points out that typical workloads, e.g., runtimes of jobs in batch systems, do not follow a uniform distribution, as the distributions are often heavy-tailed. Jain [9] states that exponential service times are commonly used. Outsterhout et al. [11] examine the performance of Spark schedulers, where the duration of spark jobs are exponentially distributed.

We also use workloads that follow an exponential distribution. An example workload is shown in Fig. 1b, where the rate parameter λ is set to 0.02, which leads to a mean work per task of 50 iterations.

From an implementation point of view, we have to be careful that the random number generator does not influence the performance results when creating OpenMP tasks. For this reason, a list of k heterogeneous task sizes are precomputed before each heterogeneous experiment and stored in a global array (we

Table 2: Experimental configurations

workload	homogeneous	heterogeneous

task strategy	`MasterTask`
	`ParallelTask`
	`ParallelFor`
		`MasterTask`
		`ParallelTask`

number of tasks n	100 000, 1 000 000	100 000, 1 000 000
work per task m	1, 1000 10 000	exp. distribution $\lambda = \{0.002, 0.02\}$
thread mapping	*compact* *scatter*	*compact* *scatter*

currently use $k = 10\,000$). When a task is spawned, we know its global task number $0 \leq i < n$, and we use the i to pick the next task size from the global list (with a wrap-around if i becomes larger than k).

Experimental Configurations We provide an overview of the experimental parameters in Table 2. For space constraints, we can only show plots for a subset of the experiments conducted. The task size parameter m is the most crucial one for comparing the OpenMP runtime systems. With $m = 1$, the time spent in each task is extremely short, and structural differences in the runtime systems get emphasized. When increasing m to 1000 or 10 000, we would like to examine whether performance differences of the runtime system can still be seen for coarser-grained tasks.

Another important parameter for efficient, multi-threaded OpenMP applications on NUMA systems is the thread-to-core mapping strategy [13]. In order to exactly implement our desired mapping strategies, we rely on the `OMP_PLACES` environment variable and define our own *compact* and *scatter* mapping strategies. We consider the exposed NUMA nodes of the system to be the basic building blocks for thread-mapping. In the *compact* strategy, we start by filling up the first NUMA node, before we map threads to the second, third NUMA node, and so forth. In the *scatter* strategy, we allocate the threads in a round-robin fashion across the NUMA nodes.

5 Experimental Results

5.1 General Experimental Factors

While experimenting with the different OpenMP runtime systems, we made two important observations. The first concerns the thread mapping strategy. In virtually all cases, the variance of the running time of our task benchmark decreases significantly if threads are pinned to specific cores. Therefore, we used thread pinning in all our experiments to reduce the number of required repetitions to obtain consistent, reproducible performance numbers.

We also noticed that the gcc compiler, especially with the `MasterTask` strategy, performed significantly inferior to its competitors. Since it uses a central

task queue, we experimented with adapting environment variables provided by OpenMP and libGOMP. We found that the wait policy had a huge positive impact on the performance of gcc, while the other compilers were unaffected. Therefore, we executed the experiments with all OpenMP runtime systems after setting the environment variable `OMP_WAIT_POLICY` to `PASSIVE`.

5.2 Homogeneous Workloads

In our first experimental analysis, we compare the performance of the different compilers on the Intel-based machine *Hydra*. Figure 2 presents strong scaling results for the case of executing $n = 100\,000$ independent tasks, each having work $m = 1$. We can observe that the `MasterTask` strategy, where the master thread creates all tasks, does not scale at all. When the number of threads increases the running time also grows, independently from the actual thread mapping strategy. Here, we can see that the larger overhead of gcc is already clearly visible starting with $p = 4$ threads.

A similar trend can be observed in the middle graphs of Fig. 2, where the `ParallelTask` version is analyzed. We can see that gcc's runtime is very unstable, as shown by the 95% confidence intervals, but gcc is very fast for 1 or 2 threads compared to clang and icc.

In the last row of this figure, we show the `ParallelFor` results as a baseline. We can observe that the task-based versions (shown in the middle row) add a significant overhead to the running time. In contrast, when applying the `ParallelFor` strategy, the parallel execution of the `add_bench` functions does show a good scaling behavior for all compilers.

In order to compare the measured runtimes in a more comprehensible way, we show the runtimes for the different compilers relative to the runtime of gcc. Thus, if a compiler has a ratio larger than 1, then this compiler was slower than gcc. On the contrary, if this ratio is below 1, the runtime of the OpenMP benchmark compiled with the respective compiler was shorter than the one compiled with gcc.

Figure 3 presents the runtime results for clang, icc, and pgcc, always relative to the runtime of gcc. In this experiment, we fixed the number of tasks to $n = 100\,000$ and used the *compact* mapping strategy. We can notice that the performance difference is relatively small if the task size m is 1000 or larger. For the task-based versions, we can observe that the other compilers outperform gcc for more than 2 threads. Yet, for two threads or less, gcc is overall the best. Interestingly, in the `ParallelFor` case, clang is clearly outperformed by both icc and gcc for virtually all thread counts.

We now turn to the AMD processors. Figure 4 presents the performance results for the 64-core machine *Nebula*. The architecture of this machine differs significantly from the Intel-based machine from before, as it has 8 NUMA nodes in total, 4 per socket. The Intel-based machine *Hydra* from the previous experiment only has two NUMA nodes, one per socket. From this figure, we can observe that the gcc compiler outperforms the competitors for small threads counts (1 and 2). However, in the `ParallelTask` case, gcc is significantly slower than the competitors when the number of cores is between 32 and 64. In the benchmarks

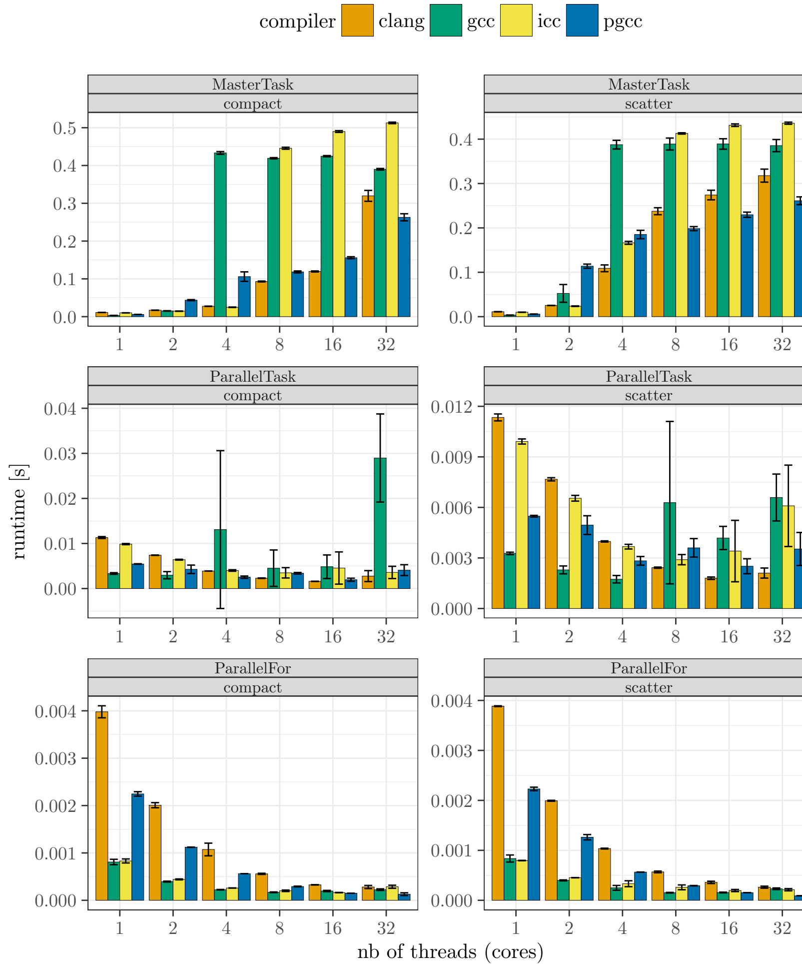


Fig. 2: Performance comparison of different compilers for $n = 100\,000$, $m = 1$; machine: *Hydra*. Error bars represent the 95% confidence interval of the mean.

on the AMD machine, the pgcc compiler provided the best overall performance, while the benchmark times obtained with clang and aocc were often slower than the ones produced by pgcc and gcc.

Last, we show the performance results for the largest shared-memory node in our experiments, which has 128 cores. In Fig. 5, we only present the results with more than 32 cores, as these cases showed the largest differences. We can observe that gcc scales very well for the `MasterTask` strategy. More interestingly, pgcc was suddenly outperformed for 96 and 128 cores for the `ParallelTask` case (middle). This was surprising as pgcc was found to be the best compiler for

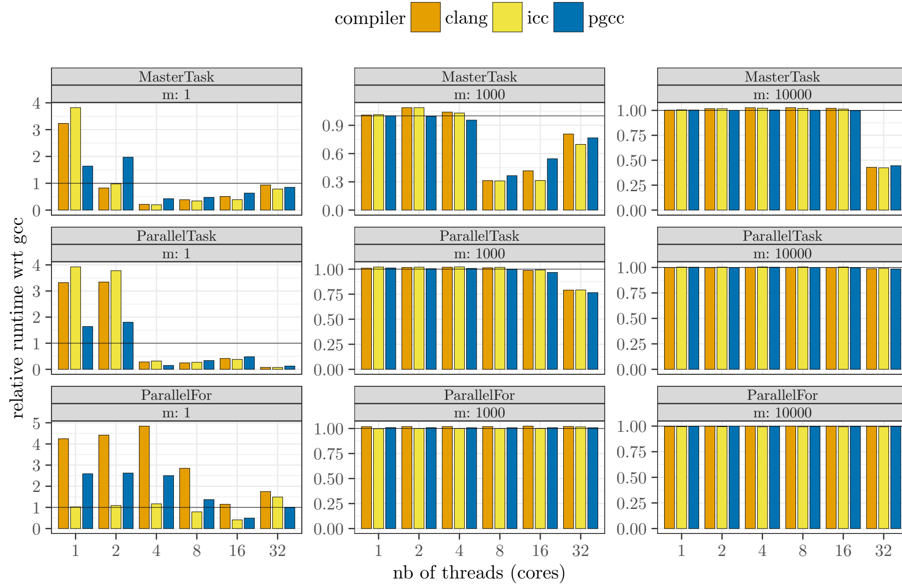


Fig. 3: Performance of compilers with respect to the runtime obtained with gcc, $n = 100\,000$, mapping: *compact*; *Hydra*.

the other machines in this case. For the `ParallelFor` case, the situation is very different, as pgcc outperforms the other compilers again significantly. We used the profiling option of our benchmark to assess how equally the tasks are balanced among the threads, in order to find the cause of the performance differences (especially for the `ParallelTask`). We could not find a correlation between the task imbalance and the running time.

5.3 Homogeneous Case: Correlation Analysis

We also wanted to assess whether the shortest running time translates to best load balanced schedule. In classic scheduling theory, a perfectly balanced schedule is a lower bound for an instance of $\mathrm{P}\,||\,C_{\max}$. However, our case is slightly different as we only have homogeneous cores, but the interconnect between the cores is heterogeneous.

In order to show the results of our study, we present one specific case that highlights our findings, which comprises $n = 100\,000$ tasks of size $m = 1$ and the `MasterTask` task creation strategy with $p = 64$ threads.

By leveraging the profiling and tracing capabilities of OMPTB, we analyzed the resulting load distribution across the different participating threads. In fact, we only used the profiling capability in this case, where the number of tasks that is executed per thread is counted. In contrast, when recording a trace, the start

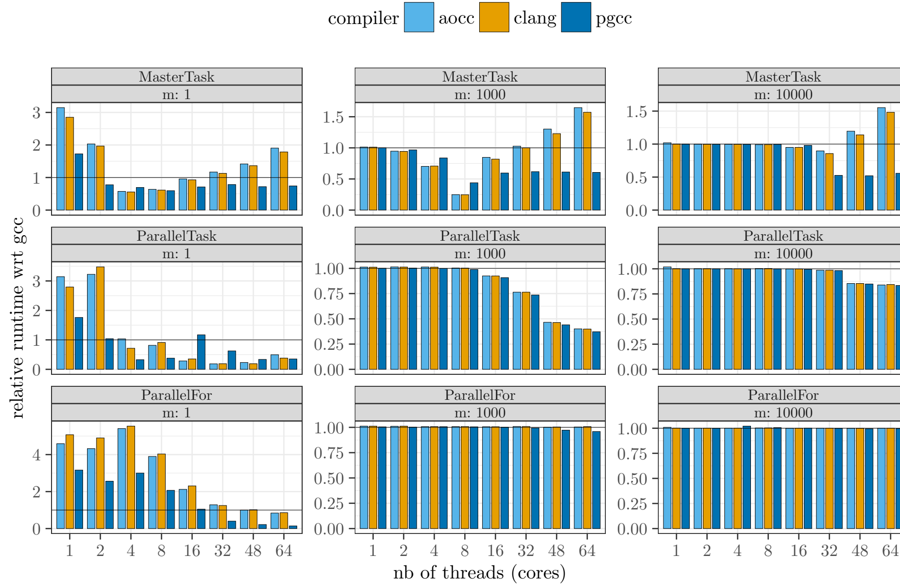


Fig. 4: Performance of compilers with respect to the runtime obtained with gcc, $n = 100\,000$, mapping: *compact*; *Nebula*.

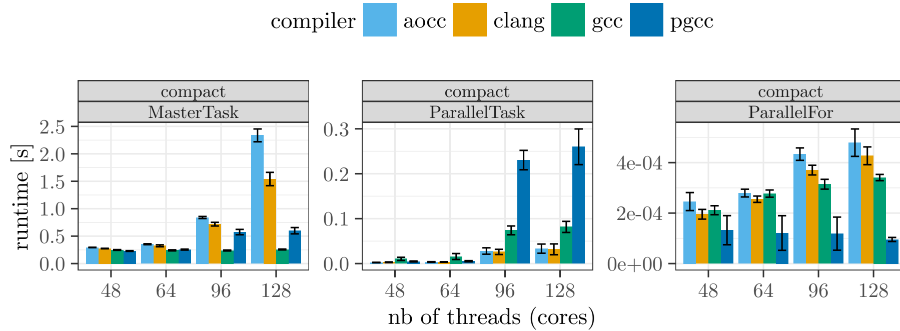
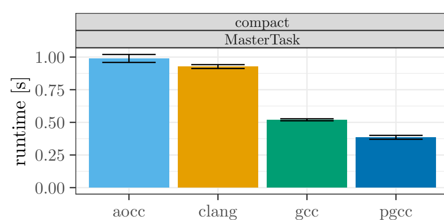


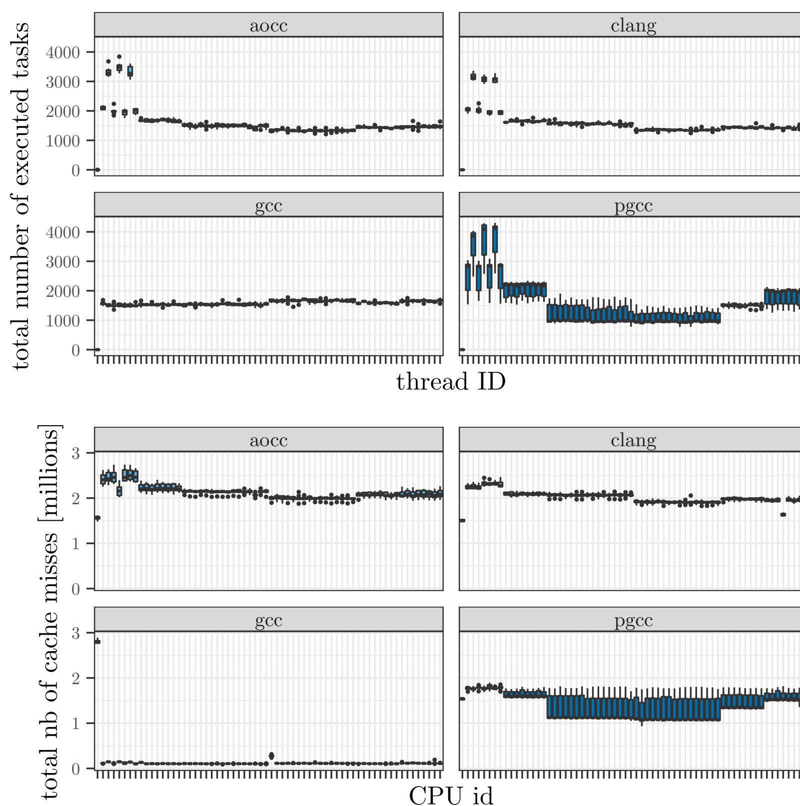
Fig. 5: Performance comparison of different compilers for $n = 100\,000$, $m = 1$ and $p > 32$; *VSC-5*. Error bars represent the 95% confidence interval of the mean.

and finish timestamp of each task will be recorded, which introduced too much overhead to the overall running time if tasks only have size $m = 1$.

Figure 6a compares the running times measured for the different compilers. In Fig. 6b, we present the number of executed tasks per thread (top) and the total number of cache misses per core. The cores on the x-axis are ordered NUMA node by NUMA node, where each NUMA node comprises 8 cores. We also ordered the



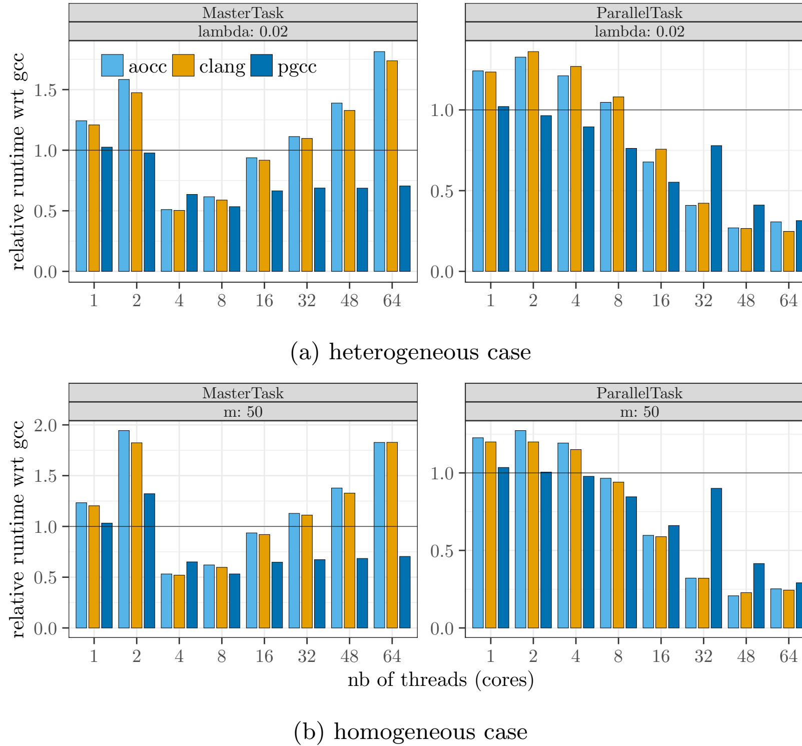
(a) Runtime comparison



(b) Executed tasks and cache misses per thread/core

Fig. 6: Compiler comparison for the specific case $n = 100\,000$, $m = 1$, $p = 64$, `MasterTask` (boxplots show results of 10 different runs), mapping: *compact*; machine: *Nebula*.

thread IDs on the x-axis in the top graph to match the core ID in the graph at the bottom. In this case, the pgcc compiler is the fastest, but we can also observe that gcc produces the most balanced load distribution across all threads. The pgcc compiler even has the largest variance in terms of number of tasks executed



(a) heterogeneous case

(b) homogeneous case

Fig. 7: Performance comparison of compilers for the heterogeneous and homogeneous workload with the same mean task size of 50, $n = 1\,000\,000$, mapping: *compact*; machine: *Nebula*.

per thread. A similar observation can be made for the cache misses. The gcc compiler produces by far the least amount of total cache misses, but it has the highest number of cache misses per core (for core 0) among all compilers. Overall, we found that neither the number of tasks per core (or a good load balancing) nor the number of cache misses is a strong predictor for the expected performance of an OpenMP runtime system. The individual implementations of data structures and locks used for the task queues also have to be taken into account.

5.4 Heterogeneous Workloads

We also investigated the scheduling performance of the different OpenMP runtime systems in the presence of heterogeneous workloads. Our initial question was whether the heterogeneity of the work done by each task fundamentally changes the performance numbers of the OpenMP runtime systems. To answer this question, we conducted experiments with two similar sets of workloads. The first workload contains heterogeneous task sizes, which are drawn from an exponential

distribution with $\lambda = 0.02$ (cf. Sect. 4). This specific exponential distribution has a mean of 50. For comparison, we also conducted experiments with a homogeneous workload, where each task size has exactly the same work, i.e., $m = 50$.

The experimental results for running these two workloads are shown in Figure 7. Although the graphs do not precisely match each other, the results for the heterogeneous case (top) are very similar to the ones obtained for the homogeneous case (bottom). We can observe that gcc is inferior for more than 16 cores for the `ParallelTask` case. However, for the `MasterTask` case, gcc outperforms both clang and aocc for many core counts. Similarly to the results shown before, the pgcc compiler offers the best overall performance for the cases considered. More importantly, the performance was mainly influenced by the number of cores (threads) and by the mean task size. The fact that the individual task sizes are distributed either homogeneously or heterogeneously only has a small impact. This finding was consistent with the other experiments that we have conducted.

6 Conclusions

We evaluated the task scheduling performance of OpenMP runtime systems found in modern compiler suites, such as gcc, icc, pgcc, or clang. We developed a benchmark called OMPTB to assess the performance of OpenMP runtime systems when processing a large number of independent tasks. In particular, we examined the scalability behavior of the runtime systems when increasing the number of cores. We also investigated the influence of the thread mapping strategy on the performance of the schedulers.

When the generated tasks have a small work, gcc is outperformed by the competitors. However, for 2 and 4 cores (threads), gcc often provides a very competitive performance. When comparing the other compilers, we observed that clang (and aocc) was often slower than icc or pgcc.

We also investigated whether the tread mapping strategy has a strong impact on the resulting performance. Here, we can give several answers. Using a thread mapping strategy improves reproducibility, as the runtime variance is significantly reduced. In our work, we examined the *compact* and the *scatter* mapping strategies. When comparing both, we cannot clearly state which one should be used, because there was no clear winner, as the better mapping strategy depends on the actual scheduling problem (e.g., number of tasks, number of cores).

We also examined the performance impact of executing heterogeneous workloads, i.e., the work of the generated OpenMP tasks differs (they have a different runtime). Surprisingly, the actual mean of the work distributions was far more important than heterogeneity, i.e., the performance numbers produced by the compilers were very similar for homogeneous and heterogeneous workloads if the mean work per task matched.

Acknowledgments

We thank Lukas Briem for helping to implement the heterogeneous workloads.

References

1. Bull, J.M., Reid, F., McDonnell, N.: A microbenchmark suite for OpenMP tasks. In: Proceedings of the 8th IWOMP. LNCS, vol. 7312, pp. 271–274. Springer (2012). https://doi.org/10.1007/978-3-642-30961-8_24
2. Chasapis, D., Casas, M., Moretó, M., Vidal, R., Ayguadé, E., Labarta, J., Valero, M.: PARSECSs: Evaluating the impact of task parallelism in the PARSEC benchmark suite. ACM Trans. Archit. Code Optim. **12**(4), 41:1–41:22 (2016). https://doi.org/10.1145/2829952
3. Clet-Ortega, J., Carribault, P., Pérache, M.: Evaluation of OpenMP task scheduling algorithms for large NUMA architectures. In: Proceedings of the Euro-Par. LNCS, vol. 8632, pp. 596–607. Springer (2014). https://doi.org/10.1007/978-3-319-09873-9_50
4. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguadé, E.: Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: Proceedings of the ICPP. pp. 124–131. IEEE Computer Society (2009). https://doi.org/10.1109/ICPP.2009.64
5. Feitelson, D.G.: Workload Modeling for Computer Systems Performance Evaluation. Cambridge University Press (2015)
6. Gautier, T., Pérez, C., Richard, J.: On the impact of OpenMP task granularity. In: Proceedings of the 14th IWOMP. LNCS, vol. 11128, pp. 205–221. Springer (2018). https://doi.org/10.1007/978-3-319-98521-3_14
7. Graham, R.L., Lawler, E.L., Lenstra, J.K., Kan, A.R.: Optimization and approximation in deterministic sequencing and scheduling: a survey. Annals of discrete mathematics **5**, 287–326 (1979)
8. Huynh, A., Helm, C., Iwasaki, S., Endo, W., Namsraijav, B., Taura, K.: TP-PARSEC: A task parallel PARSEC benchmark suite. J. Inf. Process. **27**, 211–220 (2019). https://doi.org/10.2197/ipsjjip.27.211
9. Jain, R.: The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling. Wiley (1991)
10. Olivier, S., Porterfield, A., Wheeler, K.B., Spiegel, M., Prins, J.F.: OpenMP task scheduling strategies for multicore NUMA systems. Int. J. High Perform. Comput. Appl. **26**(2), 110–124 (2012). https://doi.org/10.1177/1094342011434065
11. Ousterhout, K., Wendell, P., Zaharia, M., Stoica, I.: Sparrow: distributed, low latency scheduling. In: Proceedings of the 24th SOSP. pp. 69–84. ACM (2013). https://doi.org/10.1145/2517349.2522716
12. Schuchart, J., Nachtmann, M., Gracia, J.: Patterns for OpenMP task data dependency overhead measurements. In: Proceedings of the 13th IWOMP. LNCS, vol. 10468, pp. 156–168. Springer (2017). https://doi.org/10.1007/978-3-319-65578-9_11
13. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Assessing OpenMP tasking implementations on NUMA architectures. In: Proceedings of the 8th IWOMP. LNCS, vol. 7312, pp. 182–195. Springer (2012). https://doi.org/10.1007/978-3-642-30961-8_14
14. Yang, J., He, Q.: Scheduling parallel computations by work stealing: A survey. Int. J. Parallel Program. **46**(2), 173–197 (2018). https://doi.org/10.1007/s10766-016-0484-8
15. Zhan, X., Bao, Y., Bienia, C., Li, K.: PARSEC3.0: A multicore benchmark suite with network stacks and SPLASH-2X. SIGARCH Comput. Archit. News **44**(5), 1–16 (2016). https://doi.org/10.1145/3053277.3053279