NUMA-Aware Task Performance Analysis

Dirk Schmidl[1,2,3(✉)] and Matthias S. Müller[1,2,3]

[1] IT Center, RWTH Aachen University,
52074 Aachen, Germany
{schmidl,mueller}@itc.rwth-aachen.de
[2] Chair for High Performance Computing, RWTH Aachen University,
52074 Aachen, Germany
[3] JARA - High-Performance Computing,
Schinkelstraße 2, 52062 Aachen, Germany

Abstract. The tasking feature enriches OpenMP by a method to express parallelism in a more general way than before, as it can be applied to loops but also to recursive algorithms without the need of nested parallel regions. However, the performance of a tasking program is very much influenced by the task scheduling inside the OpenMP runtime. Especially on large NUMA systems and when tasks work on shared data structures which are split across NUMA nodes, the runtime influence is significant. For a programmer there is no easy way to examine these performance relevant decisions taken by the runtime, neither with functionality provided by OpenMP nor with external performance tools. Therefore, we will present a method based on the Score-P measurement infrastructure which allows to analyze task parallel programs on NUMA systems more deeply, allowing the user to see if tasks were executed by the creating thread or remotely on the same or a different socket. Exemplary the Intel and the GNU Compiler were used to execute the same task parallel code, where a performance difference of 8x could be observed, mainly due to task scheduling. We evaluate the presented method by investigating both execution runs and highlight the differences of the task scheduling applied.

1 Introduction

In 2007 OpenMP was extended by a new way to express parallelism through tasks. A task is an independent chunk of work in combination with an own data environment. In OpenMP tasks are executed by threads of the current team. Which thread executes which tasks is up to the OpenMP runtime and by now OpenMP offers no way for a programmer to have influence on this decision. Furthermore, there is no way for a programmer to get information about the scheduling done by the runtime.

In former work we investigated the behavior of different OpenMP runtime systems and their scheduling techniques (see [14,15] for details). It turned out, that the scheduling of tasks to threads can be extremely relevant for application performance especially on non uniform memory access (NUMA) machines.

N. Maruyama et al. (Eds.): IWOMP 2016, LNCS 9903, pp. 77–88, 2016.
DOI: 10.1007/978-3-319-45550-1_6

Furthermore, we found that a programmer can have indirect influence on the scheduling for some runtimes, like the Intel OpenMP runtime, by letting threads create tasks which they should preferably execute. This way of task creation is called `parallel-producer multiple-executor` pattern, since all threads create tasks in parallel and then also execute them in parallel.

Since in these cases the application performance is dependent on the scheduling done by the OpenMP runtime, it is highly desirable from a programmer's perspective to be able to observe the scheduling behavior in detail for performance analysis. We investigated the general ability of performance tools to analyze OpenMP task parallel programs in [13]. It turned out, that performance tools can deliver a lot of information for tasking programs, but no tool delivers an easy way to understand the impact of scheduling decisions done by the runtime on NUMA architectures.

Therefore, in this work we will present an approach to address this issue. Based on the Score-P performance measurement infrastructure [5] we implemented a method to combine a standard OTF2 event trace with hardware information of the system about memory nodes and with information about thread pinning. This allows to investigate if tasks were executed by the creating thread or if they were executed by a different thread. Furthermore we can analyze if the different thread was running on the same or a different NUMA node, which is useful information in case the task accesses data resident on the local NUMA node.

The rest of this work is structured as follows: First, we present related work in Sect. 2 before we will recap relevant information from our work on NUMA-aware task programming in Sect. 3, including an extended version of the benchmark we used to analyze the tasking behavior of different runtime systems. Then we present out method to analyze the tasking performance in a NUMA-aware manner in Sect. 4 and an evaluation is presented in Sect. 5. Finally, we draw our conclusions in Sect. 6.

2 Related Work

The concept of tasks [1] has been added in OpenMP 3.0 [9]. As was shown by Ayguadé et al. tasking is able to deliver comparable performance to OpenMP worksharing implementations [2]. This early performance comparison did not focus on multi-core multi-socket (NUMA) machines, but several others investigated this issue. Olivier et al. [8] and Broquedis et al. [3] both deal with the efficient scheduling of OpenMP tasks on NUMA systems. Our previous work [14,15] in contrast did not aim at changes to the scheduling, but tried to use the given scheduling mechanism which became common practice in OpenMP in the most efficient way on NUMA systems.

Performance analysis of task parallel programs is meanwhile possible with many different tools. We investigated the ability of a subset of these tools in [13]. Sampling based tools like the Intel VTune Amplifier [4] or the Oracle Solaris Studio Analyzer [10] can be used to gather a statistical overview of the execution

of a task parallel program. These tools allow to identify execution time in tasks and also overhead spend in the OpenMP runtime to manage tasks or idle time because no tasks are available. But, they do not allow to identify an individual task instance in the analysis and also data on the scheduling is not presented. Event based tools like the measurement system Score-P in combination with visualization tools like the profile browser Cube [11] or the event trace visualization tool Vampir [7] allow to investigate the same information than the sampling based tools, but with a higher measurement overhead for fine grained tasks. Furthermore these tools allow to identify task instances which allows for individual tasks to locate the creation and execution region in a trace file manually. With this information the scheduling can be analyzed, but for hundreds or thousands of tasks in an application run, this leads to an enormous amount of manual work. Therefore, we will present an automated method to ease this analysis in the next sections of this work.

3 NUMA-Aware Task Creation

The performance analysis techniques presented in this work are useful under different circumstances. The major requirement is, that a programmer has created the tasks in a NUMA-aware manner. What exactly is understood under `NUMA-aware` in this context is wrapped up in this section and is basically a recap of the work we presented in [14,15].

3.1 Task Scheduling in OpenMP

As mentioned before, OpenMP does not specify exactly how the runtime should schedule tasks. Even not scheduling them at all and just executing all tasks immediately would be legal according to the specification. But of course this would not lead to additional parallelism in the application and a user would be unsatisfied by the implementation. So, in the past eight years, since tasking was added to OpenMP, different scheduling mechanisms have emerged. The most relevant difference with respect to NUMA-aware scheduling is, if tasks are queued after creation in one central task queue or if they are queued in thread local task queues.

Figure 1 illustrates both approaches. On the left side a central task queue is shown. Here all threads enqueue tasks into one queue and also dequeue tasks from this data structure. On the right side of the figure thread local task queues are shown. Here, every thread has an own queue where it enqueues tasks. Every thread can also dequeue tasks from its own queue, but it can also dequeue tasks from other queues. This is then called *task stealing*. Typically stealing is only performed if the local queue is empty, since it involves more overhead.

The Intel compiler (v. 15.0) uses thread local task queues, whereas the GCC (v.4.9) compiler uses a central queue. Therefore, these compilers are used for all later experiments as representatives for one or the other approach.

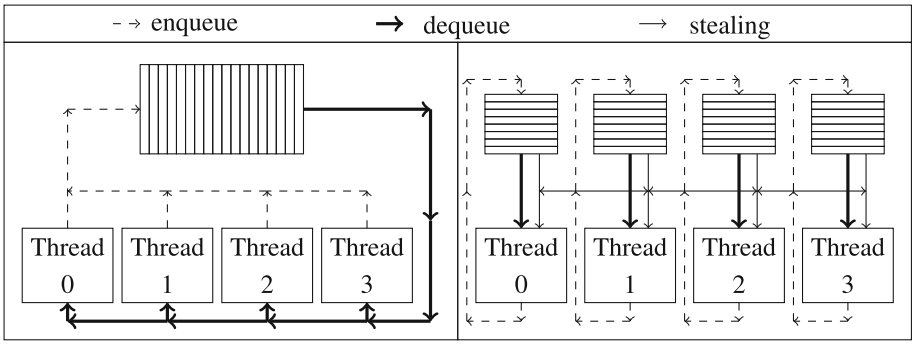


Fig. 1. Illustration of a central task queue used by four threads (left) and thread-local task queues for four threads (right).

3.2 Task Creation

NUMA-aware programming typically has to handle:

- The mapping of threads to cores on specific NUMA nodes.
- The mapping of data to NUMA nodes.
- The mapping of *work items* to threads. Work items in this context can be for example loop iterations, OpenMP sections or tasks.

The goal for NUMA-aware programming is then to execute work items by threads which run on the NUMA node where the data is located which is needed to process this work item.

Regarding the first issue, OpenMP offers support for thread pinning which allow to influence the mapping of threads to cores in any desired way. For the data mapping no support exists in OpenMP, but all common operating systems in HPC use the so called `first-touch` memory allocation policy. This policy means, that data is located on the NUMA node where it is first used. By parallel data initialization this can be used to achieve most desired mappings of data to NUMA nodes. For the mapping of work items to threads in some cases support exists, e.g. for parallel loops with a static schedule clause. But, as mentioned above, when tasks are used, the mapping of tasks to threads cannot directly be influenced by the programmer and any scheduling of tasks to threads is valid according to the specification.

In [14,15] we investigated the scheduling of tasks for different task creation patterns and for different compilers. It turned out, that the implementation of thread local task queues can be used on NUMA systems to maintain data locality to a certain amount. When the `parallel-producer multiple-executor` pattern is used, i.e. tasks are created and executed in parallel by all threads of a team, every thread fills its own local task queue. During execution all threads will then first pick tasks out of their own task queue until this queue is empty. After the queue is empty, they will start task stealing. If the programmer creates tasks by the thread which also initialized the data needed in this tasks, this results in

a situation where all threads have a task queue filled with tasks which need local data. After a thread executed all tasks of its own task queue, it will start task stealing and execute tasks on remote data, if the task is stolen from a remote NUMA node's thread. But, this situation means, that no local data needs to be processed anymore, since the local queue is empty. In this case it is better to execute remote tasks than doing nothing.

3.3 Benchmark Evaluation

To highlight the performance relevance of tasking implementations under these circumstances we used a synthetic benchmark program. The benchmark emulates a situation where:

- Many work items need to be processed which all work on separate data.
- The needed data is distributed already over the NUMA nodes of the system.
- The amount of data to process on each NUMA node is different, so there is a load imbalance problem. (If no load imbalance would exist, a loop worksharing construct with a static schedule would be preferred over tasks.)

The benchmark is designed in the following way: A set of work packages (WP) to execute is created (3840 in this case). Each WP performs a vector addition as operation, where all WPs use different vectors. The size of the vectors increases

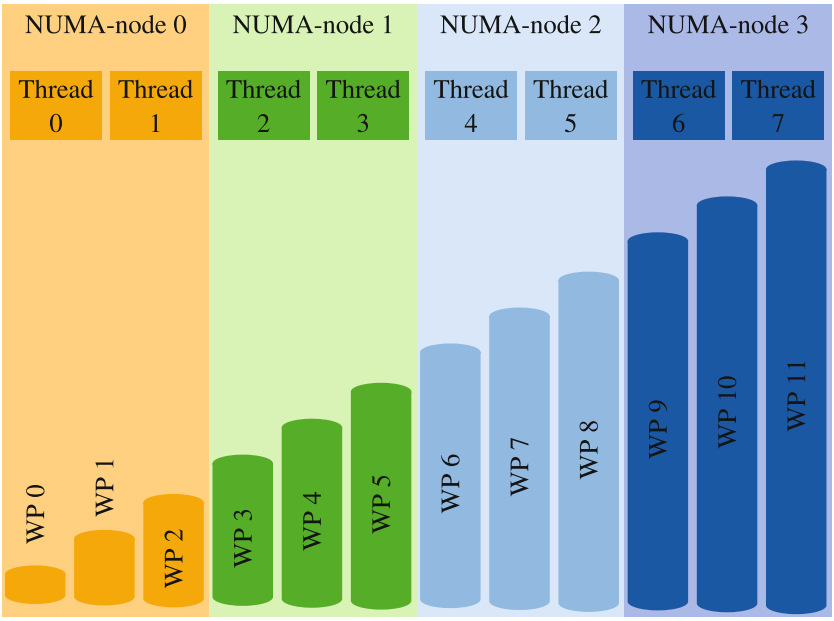
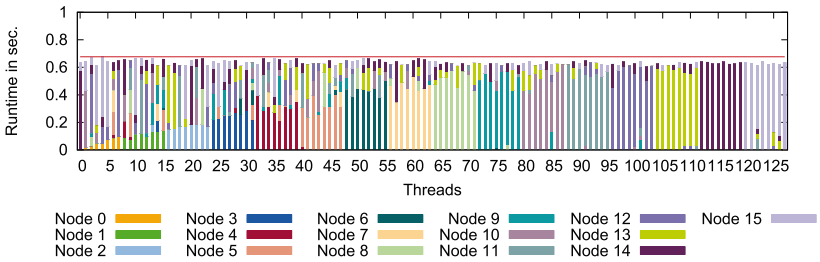


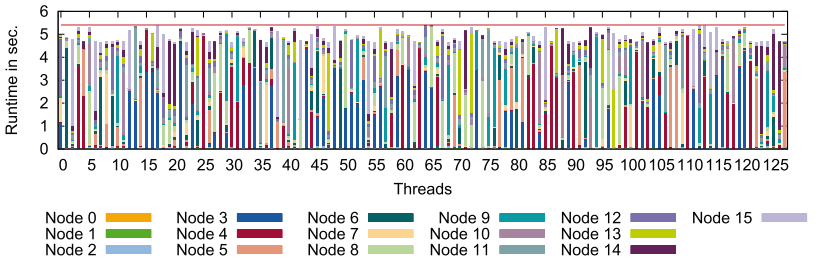
Fig. 2. Distribution of work packages (WPs) across NUMA nodes for the load balancing benchmark. Exemplary for 12 work packages (WPs) on a machine with 8 threads and 4 NUMA nodes.

linearly from the first to the last work package, resulting in a load imbalance. The vectors are distributed across the NUMA nodes evenly, i.e. the first $\frac{|WPs|}{|NUMAnodes|}$ vectors are located in NUMA node 0, the second $\frac{|WPs|}{|NUMAnodes|}$ vectors in NUMA node 1 and so on. Figure 2 illustrates the setup exemplary for a system with eight threads and four NUMA nodes and for 12 WPs to be scheduled.

During the benchmark execution the time to execute all work packages is measured individually. Furthermore we create a `firstprivate` variable for each task which is used to store the creating thread of the task. During execution of the task we use this information to find out if task stealing was applied or not.



(a) Intel Compiler



(b) GNU Compiler

Fig. 3. Load balancing benchmark results when tasking was applied. All tasks were created by the thread which also initialized the data used by the task, to maintain locality for thread-local task queues. (Color figure online)

Figure 3 shows the results of a benchmark run on a 16-socket system equipped with Intel Xeon X7550 8-core processors. In total 128 threads were executed and the tests were done with the Intel and GNU Compiler. Each bar stands for the execution time a thread executed tasks. The colors used in the bars illustrate where the data of the tasks which were executed was located. One color for every one of the 16 NUMA nodes was used. For the Intel Compiler it can be observed, that threads with a higher ID execute mainly tasks of one color, the color of the local NUMA node. Threads with a smaller ID execute less local work and steal from different other NUMA nodes. This is because they have less data to

process and after all local work was executed they start stealing tasks. When the GNU compiler is used all threads execute tasks of many different colors. This is because of the centralized task queue which cannot be used for locality aware task creation as mentioned above. The overall execution time with the GNU compiler is about 8 times higher compared to the Intel compiler. This gives evidence, that task scheduling, also it is done internally in the runtime can have a high influence on the execution of a program, so it would be good if the programmer can observe it with performance tools.

4 Task Performance Analysis

The work presented here is based on a former publication [6], where we have shown how the performance measurement system Score-P can be extended to allow event-based performance analysis of task parallel programs. This allowed us to identify different task related performance issues, like too finely or coarsely grained tasks as shown in [12]. But, it does not enable us to identify the NUMA related issues mentioned above for the following reasons.

4.1 Gathered Data

The information desirable for the analysis if tasks are executed locally or on a remote NUMA node are:

1. Information about the hardware topology, i.e. which cores are on which NUMA node.
2. Information which thread is running on which core.
3. The start and end time of every tasks execution.
4. Information which thread executed which tasks.
5. Information which thread created which tasks.

The OTF2 trace as described in [6] contains information about the start and end time of every task (3) and also information which thread executed the task (4). The second information is implicit, since the begin and end event of the task are located in the event trace of only one thread, the one which executed the task. To get information which thread created a task (5), it is necessary to have IDs for all task instances. Such IDs are not provided by OpenMP, but in [6] we presented a method to store task local IDs in a mixture of variables `private` to a task and `threadprivate` variables. These IDs are stored with every begin and end event in the OTF2 trace. Furthermore, we used the thread ID of the creating thread of a task as prefix in the task ID. This allowed us to create tasks in parallel with unique IDs without the need to synchronize between threads, as all tasks which might be created in parallel are created by different threads and thus get a different prefix for their ID. Now, we can extract this information at task begin and end events out of the task ID to obtain information (5). The information which cores belong to which NUMA node (1) of the system is static information which can be extracted before the program run from the Linux

OS. E.g. the command line tool `numactl` lists all cores of a NUMA node in a system. The information which thread is running on which core (2) must not be constant over the complete program run for all applications. If thread binding is not used at all, the OS might migrate threads at any time. If thread binding is used and different `affinity` clauses are used for different parallel regions, this also leads to changes in the mapping of threads to cores. However, in practice the majority of programs sets a fixed affinity policy, e.g. using the environment variable `OMP_PROC_BIND` and stick to this mapping for the whole application run. In such cases the mapping can easily be queried in the application and used later on for the analysis of the complete OTF2 trace file of a thread.

4.2 Data Analysis

To be able to visualize the gathered data in a user friendly way, we implemented a post processing tool to combine all gathered data in a new OTF2 trace file. This file can than be visualized in the Vampir GUI [7] for analysis.

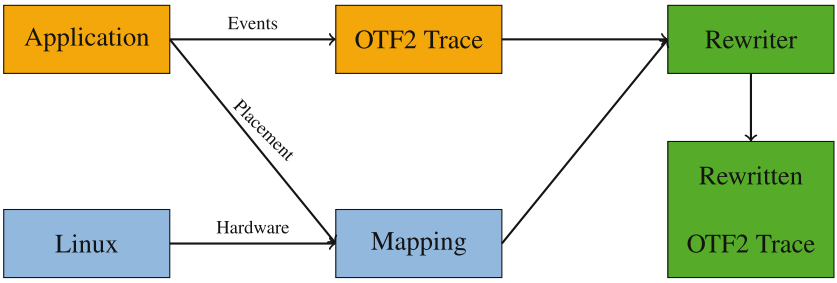


Fig. 4. Workflow of the trace rewriter tool to use hardware information to analyze NUMA related task scheduling issues.

Figure 4 illustrates the workflow for the post processing tool. During the application run the Score-P measurement infrastructure is used to generate an OTF2 trace of the application run. As usual, the trace contains `begin` and `end` events for all functions and also OpenMP events, like tasks. Every event also contains a timestamp. Also the mapping of threads to cores is written to a file during execution of the application. Furthermore hardware information on the system topology is added to this file.

After gathering this information the *Rewriter* tool reads the information on the hardware topology and the thread mapping. Then the tool reads in the original OTF2 trace and writes out a new modified trace file. If an event is not related to a task, it is copied to the new trace. If an event is a task begin event, the tool checks if the task was created by the current thread, a remote thread on the same socket or a remote thread of a different socket of the system. Furthermore, the tool adds three groups for tasks, `local`, `same socket` and `remote socket`. In the modified trace file tasks are sorted into these groups,

depending on the location of the creating thread. On a runtime system with thread local queues, this allows to distinguish if the task was stolen from a different queue or not during the analysis with Vampir later on.

5 Evaluation

Finally, we evaluate the analysis technique with the help of the benchmark presented in Sect. 3.3. We executed the benchmark once compiled with the Intel and once with the GCC compiler on a 4 socket server with 8-core Intel Xeon X7550 processors. Remember, we observed a 8x performance difference between both versions.

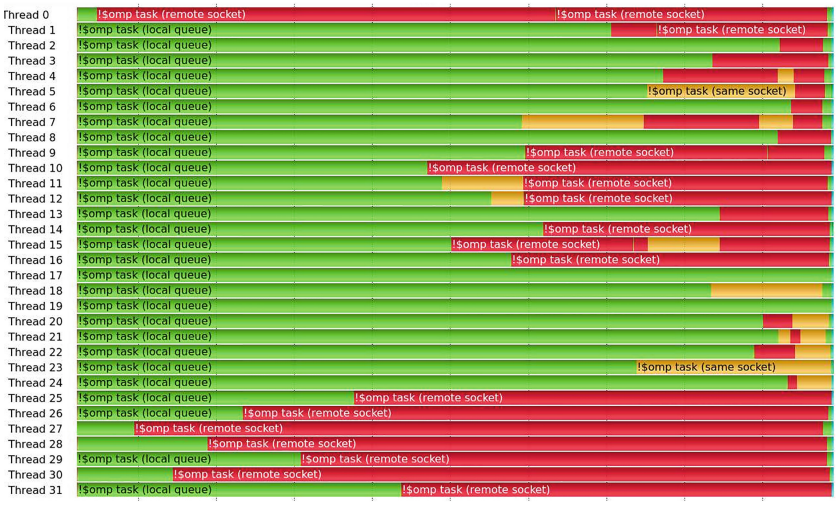


Fig. 5. Vampir screenshot showing the tasks during the execution of the load balancing benchmark with the Intel runtime. Tasks executed on the creating thread ar shown in green, stolen tasks from the same NUMA-node in orange and from remote NUMA-nodes in red. (Color figure online)

Figure 5 shows a Vampir timeline view of the execution done with the Intel compiler and Fig. 6 shows the execution using the GCC compiler. It can be observed, that the behavior with the Intel compiler is as desired. First all threads work on local (green) tasks. After some time, when no more local tasks are available, threads start doing task stealing and work on remote tasks, either on the same socket (orange) or on remote sockets (red).

With the GCC compiler and the centralized task queue, the picture is completely different. During the complete execution of the benchmark a mixture of red, orange and green tasks can be observed. But, on the very beginning, many threads also start with local (green) tasks. This is because all tasks are created

Fig. 6. Vampir screenshot showing the tasks during the execution of the load balancing benchmark with the GNU runtime. Tasks executed on the creating thread ar shown in green, stolen tasks from the same NUMA-node in orange and from remote NUMA-nodes in red. Comparing this figure to Fig. 5, a huge difference in the way tasks are scheduled in the GNU and Intel runtime can be observed. (Color figure online)

at the beginning, filling the task queue. Once the task queue exceeds a certain limit it is full and no more tasks can be queued. Then all threads execute the tasks directly instead of putting it into the queue and this of course leads to local task execution.

Overall, there are many more red tasks with the GCC compiler, which means tasks are executed by threads on different sockets than the creator of the task. Since we created the tasks with the same thread which initialized the data needed in the task, this means the executing task is also on a different socket than the data needed. So, a lot of remote memory accesses occur under these circumstances which explains the worse performance with the GNU compiler compared to the Intel compiler.

The presented technique to group the tasks in the OTF2 tracefile based on the topology information is very helpful to understand and explain performance characteristics of the benchmark program for different OpenMP runtime systems. It presents an easy to understand overview in the Vampir GUI and allow users a deeper understanding of internal task scheduling done in the OpenMP runtime. This information is helpful for application developers who want to optimize their tasking applications on NUMA systems.

6 Conclusion

Task scheduling decisions taken by the OpenMP runtime can have a high influence on the performance of task parallel programs, particularly on large NUMA

systems and when the tasks work on data already distributed across the system. We summarized previous work how NUMA aware task parallel programming can be achieved for OpenMP runtime systems working with thread local task queues. Furthermore, we presented a benchmark test to evaluate the performance difference of two runtime systems, the one from Intel and the other one from GNU as examples for runtime systems with thread local queues or a centralized queue, respectively. It turned out, that on a 16-socket NUMA machine, a performance difference of 8x was observed.

To allow users understanding such a situation in a real application code, support in performance tools is desirable. Therefore, we presented a method how standard OTF2 trace data can be combined with information on the system topology and thread placement to do an in-depth analysis of the runtime scheduling with a focus on NUMA nodes. We write topology information from the system as well as information about the thread placement into a log file. Furthermore, we measure a standard OTF2 trace with Score-P during execution. Then a post processing tool was developed to combines both information and produces a trace file where tasks are grouped in either locally executed tasks, tasks executed on the same socket as the creator and tasks executed on a remote socket. As shown during the evaluation the newly generated trace can be visualized with the Vampir GUI and contains a useful overview of the tasks executed in these different groups.

As future work, this approach could be integrated into the Score-P infrastructure directly. All information needed about the system can be gathered at the application start allowing to do the grouping at runtime. This would also allow to support situations where thread pinning is changed over runtime, e.g. by different `affinity` clauses used during execution. In such circumstances it is necessary to keep track of the thread to core mapping which is easy at runtime but hard in a post processing tool.

Acknowledgement. This work was funded by the German Federal Ministry of Research and Education (BMBF) under Grant Number 01IH13001D(Score-E).

References

1. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The Design of OpenMP Tasks. IEEE Trans. Parallel Distrib. Syst. **20**(3), 404–418 (2009)
2. Ayguadé, E., Duran, A., Hoeflinger, J.P., Massaioli, F., Teruel, X.: An experimental evaluation of the new OpenMP tasking model. In: Adve, V., Garzarán, M.J., Petersen, P. (eds.) LCPC 2007. LNCS, vol. 5234, pp. 63–77. Springer, Heidelberg (2008)
3. Broquedis, F., Furmento, N., Goglin, B., Wacrenier, P.-A., Namyst, R.: ForestGOMP: an efficient OpenMP environment for NUMA architectures. Int. J. Parallel Programm. **38**, 418–439 (2010). doi:10.1007/s10766-010-0136-3
4. Intel: Intel VTune Amplifier XE. http://software.intel.com/en-us/intel-vtune-amplifier-xe. Accessed 24 May 2016

5. Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A.D., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S.S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P - a joint performance measurement run-time infrastructure for periscope, Scalasca, TAU, and Vampir. In: Proceedings of 5th Parallel Tools Workshop, Dresden, Germany, September 2011
6. Lorenz, D., Mohr, B., Rössel, C., Schmidl, D., Wolf, F.: How to reconcile event-based performance analysis with tasking in OpenMP. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 109–121. Springer, Heidelberg (2010)
7. Nagel, W., Weber, M., Hoppe, H.-C., Solchenbach, K.: VAMPIR: visualization and analysis of MPI resources. Supercomputer **12**(1), 69–80 (1996)
8. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Prins, J.F.: Scheduling task parallelism on multi-socket multicore systems. In: Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers, ROSS 2011, pp. 49–56. ACM, New York (2011)
9. OpenMP ARB: OpenMP Application Program Interface, v. 3.0. http://www.openmp.org/mp-documents/spec30.pdf. Accessed 24 May 2016
10. Oracle: Oracle Solaris Studio 12.2: Performance Analyzer. http://docs.oracle.com/cd/E18659_01/html/821-1379/. Accessed 24 May 2016
11. Saviankou, P., Knobloch, M., Visser, A., Mohr, B.: Cube v4: from performance report explorer to performance analysis tool. Proc. Comput. Sci. **51**, 1343–1352 (2015)
12. Schmidl, D., Philippen, P., Lorenz, D., Rössel, C., Geimer, M., an Mey, D., Mohr, B., Wolf, F.: Performance analysis techniques for task-based OpenMP applications. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 196–209. Springer, Heidelberg (2012)
13. Schmidl, D., Terboven, C., an Mey, D., Müller, M.S.: Suitability of performance tools for OpenMP task-parallel programs. In: Knüpfer, A., Gracia, J., Nagel, W.E., Resch, M.M. (eds.) Tools for High Performance Computing 2013, pp. 25–37. Springer International Publishing, Basel (2013)
14. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Assessing OpenMP tasking implementations on NUMA architectures. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 182–195. Springer, Heidelberg (2012). doi:10.1007/978-3-642-30961-8_14
15. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Task-parallel programming on NUMA architectures. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 638–649. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32820-6_63