Assessing OpenMP Tasking Implementations on NUMA Architectures

Christian Terboven, Dirk Schmidl, Tim Cramer, and Dieter an Mey

JARA, RWTH Aachen University, Germany
Center for Computing and Communication
{terboven,schmidl,cramer,anmey}@rz.rwth-aachen.de

Abstract. The introduction of task-level parallelization promises to raise the level of abstraction compared to thread-centric expression of parallelism. However, tasks might exhibit poor performance on NUMA systems if locality cannot be maintained. In contrast to traditional OpenMP worksharing constructs for which threads can be bound, the behavior of tasks is much less predetermined by the OpenMP specification and implementations have a high degree of freedom implementing task scheduling.

Employing different approaches to express task-parallelism, namely the single-producer and parallel-producer patterns with different data initialization strategies, we compare the behavior and quality of OpenMP implementations with task-parallel codes on NUMA architectures. For the programmer, we propose recipies to express parallelism with tasks allowing to preserve data locality while optimizing the degree of parallelism. Our proposals are evaluated on reasonably large NUMA systems with both important application kernels as well as a real-world simulation code.

1 Introduction

The availability of cost-efficient two- and quad-socket compute nodes with large memory made non-uniform memory access (NUMA) architectures omnipresent. In a NUMA architecture, the memory is partitioned and the latency and bandwidth of memory access depend on the distance to the core from which the access occurs. The thread-centric expression of parallelism, like worksharing in OpenMP[13], works fine on such machines for well-structured code and evenly-balanced algorithms, but it often is unsuitable for recursive algorithms, unbounded loops, or irregular problems in general. Task-level parallelism provides solutions for these applications, but while threads can be bound to cores, the OpenMP specification leaves a high degree of freedom regarding the behavior of tasks to the implementation. If tasks are executed on a NUMA node remote from the data, it has to be transferred first, leading to poor performance.

In this work, we compare the behavior and quality of OpenMP tasking implementations on recent NUMA architectures of different sizes. While detailed descriptions on the inner workings of research OpenMP implementations can be found in the literature (e.g.[14] or [10]), this information is not available for commercial ones, thus we created several experiments to analyze their behavior. We observed significant differences both in the overhead of task creation as well as in the task scheduling on NUMA architectures for the four implementations from Intel, GNU, Oracle and PGI. By analyzing how the implementations execute tasks, we derived strategies for task-parallel programming that take the data allocation and work scheduling into account. For implementations that exhibit reliable and consistent behavior, we show that our strategies are successful for compute kernels as well as real-world applications.

This paper is structured as follows: the next chapter discusses related work. Chapter 3 contains our observations on how current OpenMP implementations execute tasks on NUMA architectures. In Chap. 4 we exploit this to express several compute kernels with tasks instead of employing worksharing constructs. Following in Chap. 5 we transfer our strategies to two real-world applications. Chapter 6 contains the summary of our findings.

2 Related Work

Tasking[1] has been introduced in OpenMP 3.0 and has been shown to be able to deliver comparable performance to OpenMP worksharing implementations[2]. The Barcelona OpenMP Task Suite[7] can be employed to compare the efficiency of tasking implementations for several kernels, but in contrast to this work it does not highlight differences in behavior on NUMA machines.

Several articles deal with the efficient scheduling of OpenMP tasks on multi-core multi-socket (NUMA) machines [12,3]. The main challenge is to reflect the system's memory hierarchy in the execution of the OpenMP tasks, while little or no knowledge is present of how tasks are being executed inside the application. Furthermore, task-stealing has to be applied in order to perform load balancing, which means the assignment of tasks from an overutilized thread to an underutilized thread. However, if tasks are moved to a different NUMA node, data of 'stolen' tasks remain on the NUMA node of the initialization, which then leads to remote memory accesses during task execution, as the Linux operating system with a standard kernel does not perform any auto-migration of memory pages.

3 Monitoring Task Execution

The OpenMP runtime has a lot of freedom in how to schedule tasks, providing both opportunities to optimize load balancing via 'task-stealing' and challenges to maintain data locality on NUMA architectures. Ideally tasks are distributed among the threads in a way that no thread is under- or overutilized and tasks are still close to their data, i.e. on the same NUMA node. While this goal is not achievable for any arbitrary workload and data access pattern, differences in especially the task-stealing have significant impact on the overall performance, depending on the pattern of task creation:

– *single-producer multiple-executors*: This pattern is popular for that it often requires little changes to code and data structures. The `single` construct ensures that a code region is executed by one thread only and thus avoids data races. The thread executing the `single` construct is responsible for creating all tasks of appropriate *task chunk size (tcs)* and all data necessary for the computation inside the tasks can be packed up at creation time using the `firstprivate` clause. The implicit barrier at the end of the `single` construct waits for the termination of all tasks.

– *parallel-producer multiple-executors*: A parallel OpenMP `for` worksharing construct loops over the outer iteration space with an increment specified as *task chunk size (tcs)*. In every iteration a task is spawned, performing the iteration over a range of size tcs. Thus, all threads of the team executing the worksharing construct create multiple tasks in parallel. The implicit barrier at the end of the `for` construct waits for the termination of all tasks. This pattern can also be expressed without any worksharing construct at all, as the content of a parallel region is executed by all threads of the corresponding team and thus a task construct encountered by all threads creates multiple tasks. Then the synchronization is performed at the end of the parallel region, or by appropriate task synchronization constructs or an explicit `barrier`.

Experiment Setup. We selected the Intel C/C++ 12.1.2, the Oracle Studio C/C++ 12.2 and 12.3, the GNU 4.5 and 4.6 and the PGI C/C++ 11.7-0 compilers for our comparisons, as they represent the most widely used OpenMP-enabled compilers on x86-compatible architectures. Two different machines were used to carry out our experiments:

– **4-sockets:** The bullx s6010 compute node is equipped with four Intel Xeon X7550 processors running at 2.0 GHz, thus offering 32 physical cores and 64 logical cores with hyper-threading, and 64 GB of main memory. The Intel Quickpath Interconnect (QPI) used to connect the four sockets with each other and with I/O facilities creates a system topology with four NUMA domains, with every NUMA node being separated from any other by just one hop. The system is running Scientific Linux 6.1.
– **16-sockets:** The Bull BCS system consists of four bullx s6010 systems as described above. The four systems are equipped with Bull's proprietary BCS cards providing a cache-coherent and high performant interconnect, running a single system image Scientific Linux 6.1 on 128 physical cores with 256 GB of main memory. It is important to notice that not only the BCS interconnect imposes a NUMA topology consisting of the four nodes, but still every node consists of four NUMA nodes connected via the QPI, thus this system exhibits two different levels of NUMAness.

3.1 Load Balancing vs. Data Locality

Load balancing and data locality are performance-critical aspects in shared memory parallel programming. To analyze the general behavior of OpenMP task implementations on a NUMA system, we created an artificial benchmark. It executes $128,000$ work packages, each of which reads an (inner) array with a constant value. To simulate load imbalance, the (inner) arrays differ in size: the first packages are much smaller than the last ones, so that the work is linearly increasing with steps of $128,000/n$ packages where n denotes the number of threads. The first $128,000/n$ packages consists of arrays of size $200,000/n$, the next $128,000/n$ packages read arrays of size $2 * (200,000/n)$, and so on. This benchmark allows to investigate load balancing capabilities as well as data locality effects very well: the data is distributed among the NUMA nodes using a chunk size of $128,000/n$ elements of the outer array, the first chunk of work packages reside on the NUMA node that thread 0 has been bound to, the next chunk is on NUMA node of thread 1, and so on. Thus, 'perfect' data locality would lead to weak load balance, and vice versa, a compromise has to be found. Figure 1 exemplary shows the size of work items and how they are initialized when using 8 threads.

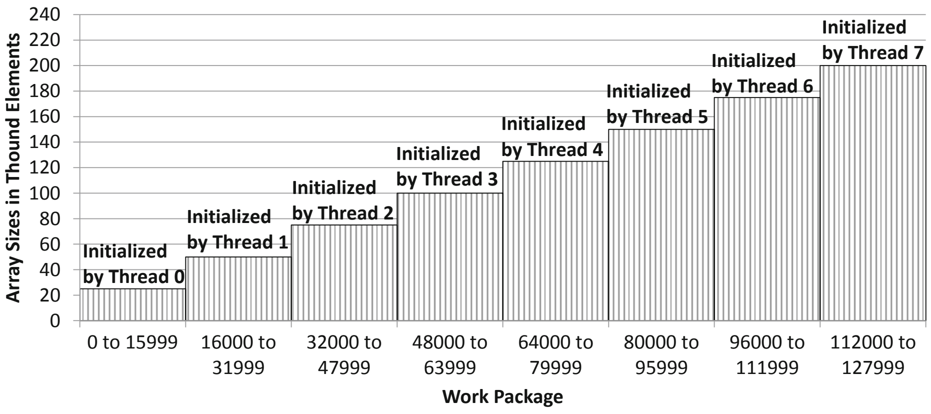


Fig. 1. Size of work packages when 8 threads are used in the load balancing experiment. Each chunk of work packages has been initialized by a different thread.

Linearly increasing load is the 'worst case' for a `for` worksharing construct with a `static` schedule and can be addressed by using a `dynamic` schedule, in which the (outer) iterations are distributed among the threads in the order in which they complete their previous work. In order to compare the behavior of the tasking implementations, we used the parallel-producer pattern along the (outer) iterations with one iteration per task. The work is measured as the total number of assignments to an inner array element. The goal is to achieve a work distribution close to 100 %, which means that every thread has to execute the same amount of work. The minimum, maximum and standard deviation of the average work per thread is shown in Table 1. Considering the 4-sockets system first, the Intel compiler distributes work almost evenly: all threads execute between 97 % and 101 % of the average work. The GNU and PGI compilers perform slightly worse, here all threads execute between 80 % and 115 % of the average work. The Oracle Studio compiler distributes work even more unbalanced over the threads, as we observed a range from 76 % to 161 %.

The load balancing on the 16-sockets machine is much worse than on the 4-sockets machine. Obviously it is harder to perform the task scheduling for 128 threads on 16 sockets than for 32 threads on 4 sockets. Thus, we expected a slightly worse result, in the order as shown by the Intel compiler: 85 % to 121 %. However, for the PGI, GNU and the Oracle Studio compilers, the distributions became extremely imbalanced. In the worst case, with the Oracle Studio Compiler, one particular thread only gets 0.09 % of the average work, whereas another thread gets 566.60 %.

Table 1. Minimum, maximum and standard deviation of the work done by a thread and percentage of local iterations for the load balancing kernel benchmark on the 4-socket and on the 16-socket machine

	4-sockets				16-sockets			
Tasking	MIN	MAX	STDV	local	MIN	MAX	STDV	local
Intel	97.65 %	100.43 %	0.51	79%	84.38 %	121.28 %	8.53	80%
GNU	81.53 %	114.15 %	5.60	80%	66.93 %	271.019 %	41.30	69%
Oracle Studio	76.02 %	161.52 %	17.68	60%	0.09 %	566.60 %	152.93	29%
PGI	83.41 %	106.56 %	5.04	82%	25.00 %	199.84 %	27.78	79%
Worksharing	MIN	MAX	STDV	local	MIN	MAX	STDV	local
Intel static	6.06 %	193.94 %	55.96	100%	1.55%	198.45%	57.29	100%
Intel dynamic	83.08 %	109.98 %	5.17	3.12%	8.61%	522.42%	148.99	0.82%

Using the same experiment again, now we shift focus on data locality. Again, we expect, that tasks are executed where they are created as long as enough work is available locally, so that there is no motivation for task-stealing. Only after all local work is complete, work from a remote location should be picked up. Table 1 also shows the results of this experiment for all investigated compilers for tasking and as well as a reference values for the Intel compiler using a `for` worksharing construct with a `dynamic` and a `static` schedule. These reference values indicate that there is an obvious trade of between load balancing and locality. The `static` schedule archives 100% locality on both systems, but the load balancing is poor. The `dynamic` schedule achieves better load balancing, but the data locality is about 3% (1%) on the 4-socket (16-socket) system. The Intel, GNU and PGI compilers achieve a local work rate of $70 - 80$ % on both machines with tasks. The Oracle compiler archives a local access rate of 60 % on the 4-sockets and 29 % only on the 16-sockets system. Although this is slightly worse than the other compilers, it is still much better than the result of the `for` worksharing loop with `dynamic` schedule. We conclude that for computations which exhibit a load imbalance and are sensitive regarding data locality, tasks offer a better alternative to traditional worksharing constructs. However, the performance depends on the task scheduling mechanisms of the OpenMP runtime, and in this experiment Intel provided the best compromise.

3.2 Task Overhead

Overhead of task construction is an important factor for the performance of OpenMP implementations. The basic measurement technique of our experiment is based on the EPCC OpenMP benchmarks [4], where the time taken for a section of sequential code is compared to the time taken for the same code enclosed in a given directive. We extended the original implementation by two new methods. In the first case only one thread generates the tasks (single-producer, left) and in the second case we create tasks with the parallel-producer pattern (right):

```
#pragma omp parallel private(j)
#pragma omp single
  for (j=0; j<innerreps* \
    omp_get_num_threads(); j++)
#pragma omp task
    delay(delaylength);
```

```
#pragma omp parallel private(j)

  for (j=0; j<innerreps; j++)

#pragma omp task
    delay(delaylength);
```

The variable `innerreps` denotes the number of repetitions and is chosen so that the execution time is significantly larger than the costs of the enclosing `single` directive. The number of generated tasks is the same for both cases and increases with the number of threads. Table 2 shows that the overhead on the 4-sockets system for all compilers is much bigger for the single-producer (*sin-pro*) pattern than the for the parallel-producer (*par-pro*) pattern: for single-producer the overhead increases with the number of tasks. While the PGI runtime has a maximum overhead of approximately $58\ \mu s$, the GNU runtime levels out at more than $1,100\ \mu s$. Compared to that, the parallel-producer pattern incurs much less overhead. Again the Intel and PGI runtime ($0.3\ \mu s$ and $3.3\ \mu s$ with 32 threads, respectively) deliver outstanding results. The overhead increase with the GNU an the Oracle Studio compilers is much more moderate compared to the single-producer pattern, but still an order of magnitude higher than for the other two runtime implementations.

The experiments on the 16-sockets machine presented in Table 3 show the same trends concerning the two patterns. However, it also shows that with the single-producer pattern the absolute overhead rises sharply with 128 threads for all implementations and with the GNU and Oracle Studio runtime for the parallel-producer pattern as well. The overhead of the task generation for the single-producer with the GNU compiler is more than $40,000\ \mu s$ while it is less than $900\ \mu s$ with the PGI compiler. In summary the Intel runtime generates tasks with least overhead of only $1.7\ \mu s$ (128 threads) using the parallel-producer pattern.

Table 2. Overhead of task creation on 4-sockets in μs

	Threads	1	2	4	8	12	16	24	32
Intel	sin-pro	0.23	1.20	1.36	1.75	59.05	156.12	560.40	764.85
	par-pro	0.11	0.19	0.36	0.24	0.27	0.17	0.30	0.26
GNU	sin-pro	2.21	2.04	6.88	83.66	185.04	304.44	652.26	1126.18
	par-pro	1.86	2.09	2.92	5.86	10.11	14.28	27.05	44.22
ORACLE	sin-pro	0.09	1.05	9.24	59.89	139.49	211.27	299.22	424.22
STUDIO	par-pro	0.09	1.32	3.43	4.33	8.47	14.06	18.55	39.65
PGI	sin-pro	0.03	2.98	2.66	2.69	4.43	9.36	34.22	57.79
	par-pro	0.03	1.26	1.37	0.95	1.69	1.90	2.49	3.26

4 Task Behavior on NUMA Architectures

In this chapter we exploit the insights gathered in the previous one to create task-parallel implementations of two compute kernels, namely STREAM[11] and a

Table 3. Overhead of task creation on 16-sockets in μs

	Threads	1	2	4	8	16	32	64	128
Intel	sin-pro	0.12	1.89	187.68	199.57	492.64	2440.62	2432.05	5656.16
	par-pro	0.12	0.32	0.93	0.17	0.72	1.11	1.32	1.69
GNU	sin-pro	1.75	1.80	14.57	209.79	785.69	1361.50	11938.48	40555.73
	par-pro	1.74	1.98	2.89	34.76	324.20	401.80	1870.33	9908.19
ORACLE	sin-pro	0.09	3.52	37.02	176.28	643.73	1534.76	3571.80	7440.76
STUDIO	par-pro	0.09	1.89	5.95	18.42	88.01	235.51	505.06	1183.10
PGI	sin-pro	0.69	5.25	4.77	14.10	62.57	160.95	367.81	892.55
	par-pro	0.02	2.68	2.35	3.45	6.94	23.58	51.35	357.38

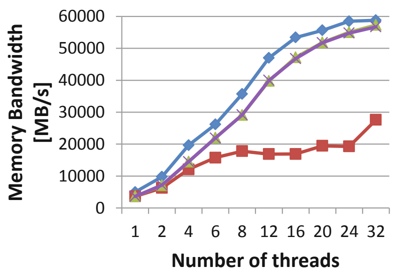
Sparse-Matrix-Vector-Multiplication in a CG-method[9], which both are very sensitive regarding the memory access pattern.

4.1 STREAM

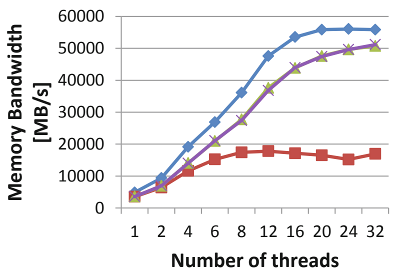
For the sake of brevity we only examine results from the triad operation, they are consistent with the other ones. Figure 2 shows the results for the Intel, Oracle and GNU compilers only, as the PGI compiler failed to compile our experiment framework correctly (Internal error: assertion failed). The arrays have a dimension of $256,435,456$ `double` elements, which results in 1.96 GB of memory consumption per array, or 5.87 GB of total kernel size in the triad operation. This kernel size is much larger than the accumulated cache size and thus we achieve reliable measurements of the memory bandwidth of the system.

Considering the 4-sockets machine first, all three compilers deliver roughly the same performance for the traditional worksharing-based parallelization, which we refer to as *workshare: static-init for-loop* and regard as a reference. In this variant, a `static` schedule is employed both during data initialization and the actual computation, meaning that for t threads the arrays are divided into t parts of approximately equal size. Given four NUMA nodes in the system and a *scatter* thread binding, meaning threads are spread as far apart as possible, $\frac{t}{4}$ threads will be bound to each NUMA node, resulting in an even data distribution over all NUMA nodes in the system. We compared this to the following task-parallel variants, for which we found a task chunk size of $65,536$ iterations per task to be optimal, although it does not have a significant influence on the performance as long as enough tasks are spawned to generate enough parallelism and as long as the work per task is computationally expensive enough compared to the task creation and scheduling overhead:

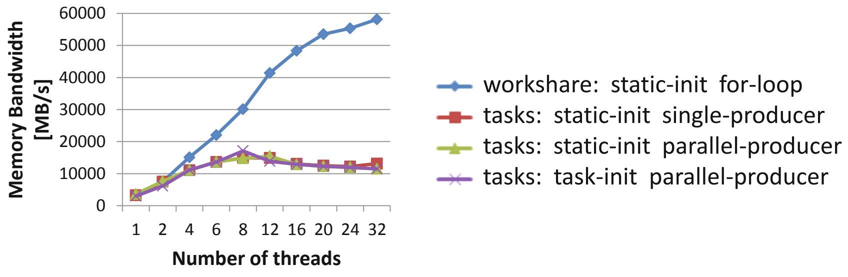
- *tasks: static-init single-producer*: The data initialization is performed in the same way as in the original parallel version. The generation of tasks is performed by one thread only (*single-producer multiple-executors* pattern).
- *tasks: static-init parallel-producer*: Again the data initialization is performed in the same way as in the original parallel version, but now the creation of tasks is performed in parallel (*parallel-producer multiple-executors* pattern).



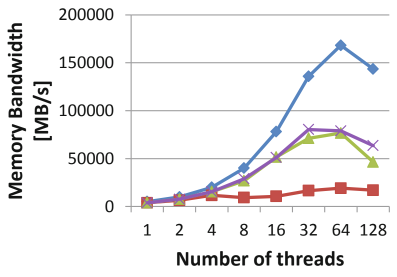
(a) triad on 4S: Intel



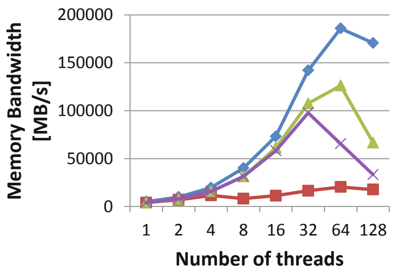
(b) triad on 4S: Oracle



(c) triad on 4S: GNU



(e) triad on 16S: Oracle



(f) triad on 16S: Intel

Fig. 2. STREAM triad operation on the 4-sockets and 16-sockets system

– *tasks: task-init parallel-producer*: The data initialization and the computation is performed task-parallel by applying the same pattern to both code regions.

Still examining the 4-sockets (4S) machine in Fig. 2(a)-(c), for the Intel and the Oracle compiler, the worksharing version outperforms the best task-parallel version by just 3 % to 5 %. Regarding these runtimes, the two task-parallel variants employing the parallel-producer pattern deliver approximately the same performance, as both distribute the data in a nearly optimal fashion over the NUMA nodes. If the parallel-producer pattern is used in the data initialization and the computation, the OpenMP runtimes of Intel and Oracle are able to maintain data affinity. However, this does not work with the GNU compiler, for which all three tasking variants deliver about the same performance - the base performance that can be achieved on this machine for random memory access. For all three compilers, the single-producer tasking version clearly suffers from two effects: (1) the runtime cannot maintain data affinity, as all tasks are created from a single NUMA node and the work-stealing will just pick arbitrary tasks from the queue; and (2) the single thread responsible for creating the tasks cannot completely keep the other threads executing the tasks busy. If only one thread creates all the tasks, the runtime's task-stealing mechanism cannot take the data distribution into account during the 'stealing' and thus the performance on NUMA systems obviously suffers.

The situation looks different on the 16-socket (16S) machine, where the peak performance of the task-parallel versions is significantly below the `for` workshare version with a `static` schedule. Interpreting comments from the Intel OpenMP runtime, the system topology is assumed to consist of 16 packages (= processor sockets), the two levels of NUMAness are not respected. Corresponding to the observations in Chap. 3.1, with higher numbers of threads the task-to-thread affinity is not 'strong' enough to prevent disadvantageous task-stealing. The single-producer variant is far behind, as the 4S results already implied. Futhermore, employing 128 threads on that machine is not profitable, similar to experiences made with other big SMPs in the past. For the final paper we will investigate measurements with 120 or 124 threads.

4.2 SMXV in a CG Kernel

While STREAM served our purpose as a benchmark indicating fine differences in the memory access pattern, the Sparse-Matrix-Vector-Multiplication (SMXV) in a CG-Method [9] much more resembles a real-world compute kernel as part of many PDE solvers. Depending on the problem the matrix for the system of linear equations can be very irregular. In this case the sparse matrix vector product is a typical example of the importance of adequate load balancing. Especially in cases where the optimal work distribution cannot be calculated in advance, we expect task-parallel implementations to help avoiding performance issues. On the one hand the programmer has to ensure that a sufficient number of tasks is used to avoid load imbalance, on the other hand too many tasks introduce additional overhead. In our CG implementation all vector operations and the dot-product are parallelized with OpenMP `for` constructs. Only the SMXV is parallelized with tasks. The work is distributed by chunks of rows and the chunk size is the same for each task, calculated as

$$chunk\_size(tasks) = \begin{cases} \lfloor N/tasks \rfloor, & \text{if } N\%tasks = 0 \\ \lfloor N/tasks \rfloor + 1, & \text{otherwise} \end{cases} \tag{1}$$

where N is the dimension of the square matrix and $tasks$ the number of tasks. The matrix used here represents a computational fluid dynamics problem (Fluorem/HV15R) and is taken from the University of Florida Sparse Matrix Collection [5]. The dimension is $N = 2,017,169$ and the number of nonzero values is $nnz = 283,073,458$, which results in a memory footprint of approximately 3.2 GB, so that the data set is big enough to not fit into the caches, even on the 16-sockets machine. The data is initialized using a `for` worksharing construct with a `static` schedule.

Figure 3 shows the performance of the SMXV when executing 1000 CG iterations. We compare the performance of the different OpenMP implementations on both machines. In almost all cases the Intel compiler delivers the best performances. For both machine types the parallel-producer pattern reaches a significantly higher performance than the single-producer pattern when using the Intel or the Oracle Studio implementation. In contrast, the peak performance achieved with the GNU compiler is below 9 GFLOPS (see 3(a)/3(b)) or rather below 5 GFLOPS (see 3(c)/3(d)) independent of the pattern. The figure also shows that even with more than $100,000$ tasks the performance of the Intel compiler for the parallel-producer variant is stable in contrast to the single-producer pattern. The behavior for the other compilers is similar in this point, although the performance decrease becomes visible with lower amount of tasks already.

Figure 3(c) shows that all implementations do not scale on the 16-sockets system when the tasks are created by one single thread. In contrast to that the Intel compiler reaches up to 21 GFLOPS on the same system (see 3(d)) in the parallel-producer variant. Task-stealing done by the OpenMP runtime to perform load balancing by the assignment of tasks from an overutilized to an underutilized thread. Table 4 shows the percentage of tasks which are executed by a different thread than it was created from. As expected, for the single-producer pattern more than 90 % of the tasks are not executed by the same thread. Furthermore, for the parallel-producer pattern only 2.9 % or rather 8.6 % of the tasks are executed by a thread which did not create this task. This means that the amount of remote data accesses introduced due to task-stealing is very low, which results in much better performance. However, Table 4 also shows that there is no difference for the GNU runtime in all cases, meaning that this kernel does not benefit from the parallel producer pattern. The Oracle Studio runtime is almost as good as the Intel runtime on the 4-sockets system, but the amount of remote accesses increases to over 40 % on the 16-socket system.

Table 4. Percentage of remote data accesses for the single- and parallel-producer pattern on the 4-sockets and 16-sockets systems using the CG kernel with 1024 tasks

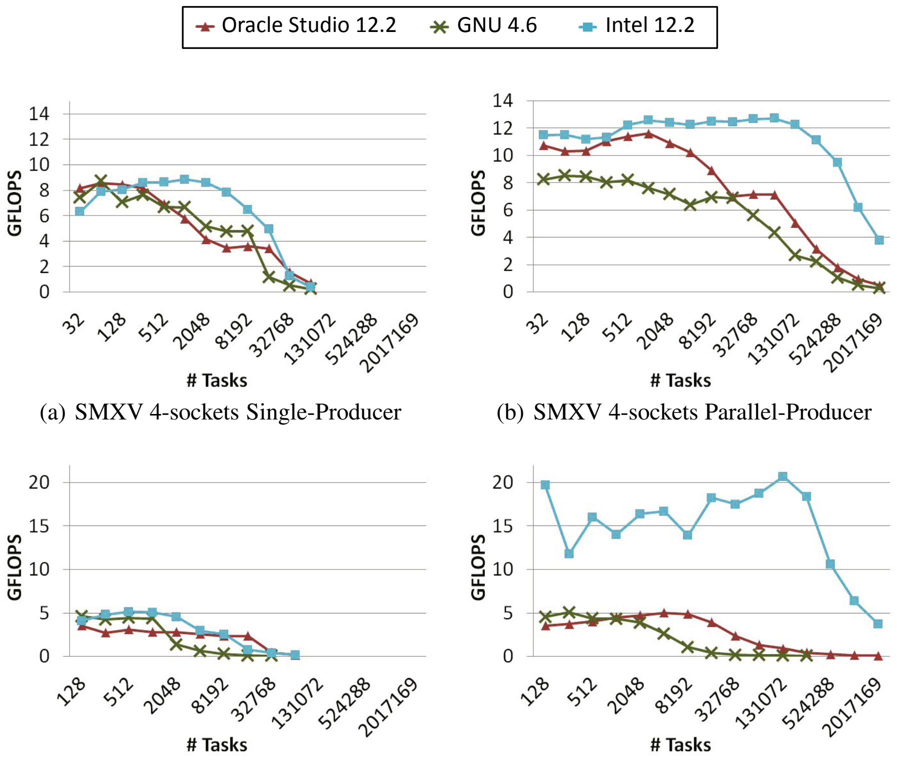
	4-sockets		16-sockets	
	single	parallel	single	parallel
Intel	96.21 %	2.87 %	99.22 %	8.61 %
GNU	96.87 %	96.90 %	99.04%	99.14 %
ORACLE STUDIO	95.97 %	4.04 %	98.24 %	41.02 %

5 Application Case Studies

Finally, we compare the different compilers for two real-world applications. Both codes have been parallelized with nested parallel regions and we added a new version utilizing OpenMP tasks. In both versions the parallelism is expressed in exactly the same way.

FIRE: The Flexible Image Retrieval Engine (FIRE) [6] was developed at the Human Language Technology and Pattern Recognition Group[1] of RWTH Aachen University. The retrieval engine takes a set of query images and for each query image it returns a number of similar images from an image database.

[1] http://www-i6.informatik.rwth-aachen.de



(a) SMXV 4-sockets Single-Producer (b) SMXV 4-sockets Parallel-Producer

(c) SMXV 16-sockets Single-Producer (d) SMXV 16-sockets Parallel-Producer

Fig. 3. Performance of SMXV for different Implementations

NestedCP: NestedCP [8] is developed at the Virtual Reality Group of the RWTH Aachen University[2] and is used to extract critical points in unsteady flow field datasets. Critical points are essential parts of the velocity field topologies and extracting them helps to interactively visualize the data in virtual environments.

Figure 4 shows the runtime and speedup of the FIRE and the NestedCP codes on the 16-sockets machine comparing the tasking version to the one with nested parallel regions. Only the Intel, GNU and Oracle Studio compilers have been investigated, as the PGI compiler failed to compile any of the two codes successfully.

Two observations are important for our discussion. Firstly, the best results for both codes are achieved using the tasking version with the Intel compiler. For the FIRE code a speedup of 127 is reached and for NestedCP a speedup of about 33, both on 128 cores. This version outperforms in both cases the nested parallel version, if the machine is fully utilized. This fact shows that the tasking paradigm works well for both applications and that the superior load balancing behavior of tasks compared to parallel regions can improve the programs performance.

[2] `http://www.vr.rwth-aachen.de`

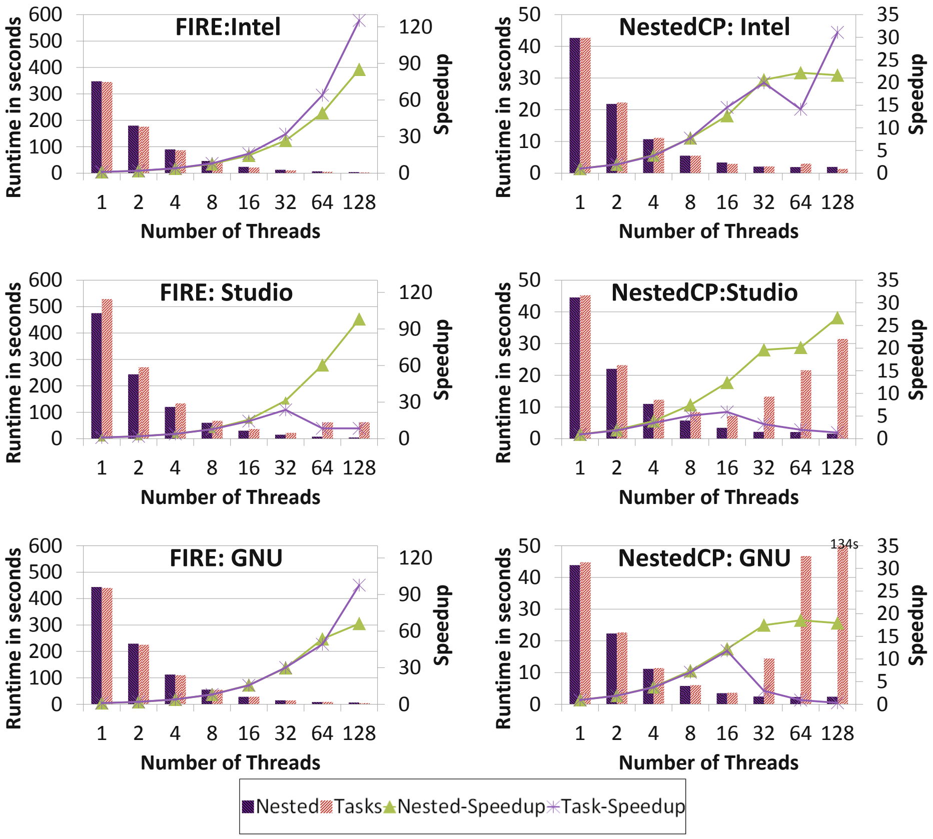


Fig. 4. Performance of the FIRE and NestedCP codes. A task-parallel version is compared to a nested variant.

Secondly, the behavior differs a lot between the compilers. With the GNU and Oracle Studio compilers, NestedCP does not scale to more than 16 threads at all with the tasking version. The FIRE code scales up to a speedup of 100 with the GNU compiler, with the Oracle compiler the performance drops down, when more than 32 threads are used. However, the FIRE version using nested parallel regions delivers the best speedup using the Oracle Studio compiler. These differences in the performance behavior of compilers and/or runtimes makes it nearly impossible to write code that performs equally well on a variety of platforms, meaning different hardware architectures and different OpenMP implementations.

6 Summary

The introduction of task-level parallelism in OpenMP raised the level of abstraction compared to thread-centric worksharing models, by delegating the responsibility of distributing the work among the threads to the runtime. On hierarchical NUMA architectures, tasks might exhibit poor performance if remote data is accessed frequently, that means if the runtime cannot maintain data locality when selecting a thread to execute a given task. If the system topology is not too complex, and if thread binding is used and the task-parallelism is expressed using an appropriate pattern, such as parallel-producer, OpenMP runtimes can maintain data affinity and thus achieve performance on par with or even better than state-of-the-art worksharing implementations. Comparing the OpenMP implementations on the 4-socket system, particularly the Intel runtime showed consistent behavior and incurs little overhead in task creation.

However, there were significant differences in behavior between the four OpenMP implementations on the 4-sockets machine and especially on the 16-sockets machines. In all kernels with all implementations the performance did not increase in the same way as the hardware's capabilties.

If the behavior of an OpenMP runtime differs a lot from another one, application performance gets hurt. Furthermore, the expectations from observation on the 4-sockets machine were not applicable on the 16-sockets machine, because the complexer topology was mostly not correctly respected. If a weak implementation does not offer reliable behavior, this also weakens the attractivity of the OpenMP tasking programming model.

Acknowledgement. Parts of this work were funded by the German Federal Ministry of Research and Education (BMBF) under Grant No. 01IH11006.

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 Transactions on Parallel and Distributed Systems 20(3), 404–418 (2009)
2. Ayguadé, E., Duran, A., Hoeflinger, J., 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. International Journal of Parallel Programming 38, 418–439 (2010) 10.1007/s10766-010-0136-3
4. Bull, J.M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In: Proceedings of First European Workshop on OpenMP, pp. 99–105 (1999)
5. Davis, T.A.: University of Florida Sparse Matrix Collection. NA Digest, 92 (1994)
6. Deselaers, T., Keysers, D., Ney, H.: Features for image retrieval: an experimental comparison. Information Retrieval 11(2), 77–107 (2008)
7. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In: Parallel Processing, (ICPP 2009), pp. 124–131 (September 2009)
8. Gerndt, A., Sarholz, S., Wolter, M., Mey, D.A., Bischof, C., Kuhlen, T.: Nested OpenMP for Efficient Computation of 3D Critical Points in Multi-Block CFD Datasets. In: Proceedings of the ACM/IEEE, SC 2006 Conference, p. 46 (November 2006)
9. Hestenes, M.R., Stiefel, E.: Methods of Conjugate Gradients for Solving Linear Systems. Journal of Research of the National Bureau of Standards 49(6), 409–436 (1952)

10. LaGrone, J., Aribuki, A., Addison, C., Chapman, B.: A Runtime Implementation of OpenMP Tasks. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 165–178. Springer, Heidelberg (2011)
11. McCalpin, J.: STREAM: Sustainable Memory Bandwidth in High Performance Computers (1999), `http://www.cs.virginia.edu/stream` (accessed March 29, 2012)
12. 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)
13. OpenMP ARB. OpenMP Application Program Interface, v. 3.1, `http://www.openmp.org`
14. Teruel, X., Martorell, X., Duran, A., Ferrer, R., Ayguadé, E.: Support for OpenMP tasks in Nanos v4. In: Lyons, K.A., Couturier, C. (eds.) Proceedings of the 2007 Conference of the Centre for Advanced Studies on Collaborative Research, pp. 256–259. IBM (October 2007)