Task-Parallel Programming 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 multicore era has led to a renaissance of shared memory parallel programming models. Moreover, the introduction of task-level parallelization raises the level of abstraction compared to thread-centric expression of parallelism. However, tasks might exhibit poor performance on NUMA systems if locality cannot be controlled and non-local data is accessed.

This work investigates various approaches to express task-parallelism using the OpenMP tasking model, from a programmer's point of view. We describe and compare task creation strategies and devise methods to preserve locality on NUMA architectures while optimizing the degree of parallelism. Our proposals are evaluated on reasonably large NUMA systems with both important application kernels as well as real-world simulation codes.

1 Introduction

Recent multi-core architectures and the availability of cost-efficient two- and quad-socket compute nodes with large memory led to an increasing interest in shared memory programming models, both in combination with MPI or as the sole source of parallelism. The increasing number of cores imply a non-uniform memory access (NUMA) to provide appropriate memory bandwidth, even on commodity x86 architectures. In a NUMA architecture, the memory is partitioned and the latency and bandwidth of a memory access depend on the distance to the core from which the access occurs. The thread-centric expression of parallelism, like worksharing in OpenMP, works fine on such machines for well-structured code and evenly balanced algorithms. However, this has been found unsuitable to be applied to certain types of codes, such as recursive algorithms, unbounded loops, or irregular problems in general. Task-level parallelism provides solutions for these applications and promises to provide a high level abstraction for the programmer.

While threads can be bound to cores, or to a subset of the machine in general, recent parallelization paradigms embracing tasks do not offer means to control by which thread and on which core tasks are executed. Thus, they might exhibit poor performance on NUMA systems if tasks are executed on a NUMA node that does not contain the data being consumed during execution, and non-local data has to be accessed. As OpenMP [11] has become the de-facto standard for shared memory parallelization in

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

HPC applications, we concentrate on the OpenMP tasking model in this work. By observing how current implementations work and execute tasks, we derive strategies for task-parallel programming that take the data allocation on NUMA architectures into account. We show that our patterns are successful for real-world applications by comparing task-parallel and thread-centric implementations on current NUMA systems.

This paper is structured as follows: Chap. 2 summarizes related work. Chap. 3 first emphasizes our expectations on the tasking model compared to a thread-centric view of parallelism and then explains the various patterns to express parallelism with tasks. In Chap. 4 we discuss our observations on the behavior of task-parallel kernels on NUMA architectures. In Chap. 5 we report our findings applying the presented patterns to real-world applications. Chap. 6 contains our conclusions and advice to programmers.

2 Related Work

Tasking [2] has been introduced in OpenMP 3.0 to allow for the expression and exploitation of unstructured parallelism. In order to extend the OpenMP standard, a reference implementation has to be provided for every new feature, which in the case of tasking was done at the Barcelona Supercomputing Center [15]. It was shown early on that OpenMP tasking is able to deliver comparable performance to OpenMP worksharing implementations [3]. While we also compare task-parallel implementations to worksharing, we additionally focus on the programmer's view of how to express task-parallelism especially on NUMA architectures. We introduce patterns for data setup/initialization as well as the actual computation and carry our findings from kernels to real-world application codes.

Several articles deal with the efficient scheduling of OpenMP tasks on multi-core multi-socket (NUMA) machines [10,4]. 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 task 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 does not perform any auto-migration of memory pages. In our work, we exploit knowledge of the implementation internals to present task generation patterns that allow task scheduling while maintaining data locality.

3 Patterns for Task-Parallelism

Two performance-critical aspects of shared memory parallel programming are load balancing and data locality. While the execution of iterations in loop-level parallelization can be controlled by schedule clauses and the threads in the team executing the worksharing construct can be bound to cores, the behavior of tasks is much less predetermined by the OpenMP specification. It is specified that a task may be picked up by any thread of the current team and that `tied` tasks may be suspended at so-called task scheduling points, `untied` tasks at every point in time. These restrictions allow

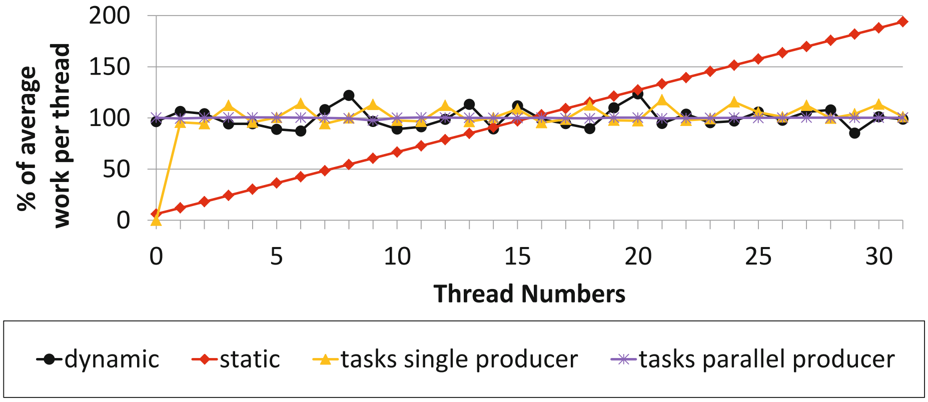


Fig. 1. Distribution of loop iterations to threads with linearly increasing load

OpenMP implementors to schedule tasks in many different ways. For example, executing all tasks by the creating thread immediately after encountering the task construct would fullfil the specification, but result in sequential execution. Pushing tasks to multiple task queues and applying work stealing approaches is a similarly valid way to handle tasks. The user has no direct influence on the scheduling decisions of the OpenMP runtime, but the scheduling has significant influence on the efficiency of a task-parallel program.

In general, there are two different patterns 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 contructs or an explicit `barrier`.

Load Balancing. In order to investigate the load balancing capabilities of tasks, we created a simple test program and used the Intel C/C++ Compiler version 12.1.2. In this program, a loop over an array with 128,000 elements has been parallelized with a `for` worksharing construct and also with two task-parallel approaches for a direct comparison. In every iteration an array element is filled with a constant value. To investigate the load balancing behavior, every array is an array itself, and the length of the inner arrays is increasing linearly. In our experiments we record which thread performed the work on the elements of the outer array, resulting in a mapping of work to threads. Fig. 1 shows the distribution of iterations for the parallel `for` workshare variant with `static` and `dynamic` loop schedules as well as for the task-parallel executions. All measurements were carried out using 32 threads on the system described in Chap. 4, consisting of four NUMA nodes with 32 physical cores in total. As expected, with a `static` schedule the work is distributed unevenly over the threads, resulting in load imbalance. The linearly increasing load per iteration is the 'worst case' situation for this schedule. It assigns $\frac{128{,}000\ its}{32\ threads} = 4000$ iterations to every thread in which the first iterations are computationally much less expensive than the last chunk of iterations. The `dynamic` schedule distributes the load much better over all threads, which then execute between 87% and 120% of the average work. In the task-parallel single-producer version, the first thread executes nearly no work, since it is responsible for creating all the tasks. But the distribution of work to the other 31 threads is as good as in the `dynamic` schedule variant. The parallel-producer scheme reaches a nearly even distribution of work over all threads, close to the optimal load balancing.

Data Affinity. We also aim to understand the behavior of worksharing and task constructs with regard to data locality, taking the same array of arrays as described above for our experiment setup. The data has been distributed among the NUMA nodes using a chunk size of 4000 elements of the outer array, meaning the first 4000 elements (including the inner arrays) reside on the NUMA node that thread 0 has been bound to. The next chunk is on the NUMA node of thread 1, and so on. Again we recorded for every thread the number of iterations it worked locally and remotely, the averages work (number of updates of an inner array element) are shown in Table 1. For the `static` schedule only local accesses occur since the distribution of array elements among the threads is exactly the same for both the initialization and the iteration phases. The `dynamic` schedule and the task-parallel single-producer scheme, which both show good load balancing, lead to only about 3% of local accesses, because for both the distribution of iterations or task, respectively, to threads is undeterministic and obviously is not the same for the initialization and iteration phases. The parallel-producer scheme achieves a local access rate of about 80%. Note that due to the uneven distribution of data over NUMA nodes, 'perfect' data locality would imply weak load balancing. The parallel-producer pattern delivers the best compromise between load balancing and locality.

We also investigated how to employ tasks in the initialization of a sparse matrix structure, as it appears for example in CG-type solvers, comparing to the common practice of initializing the data in a parallel loop over the matrix rows [14]. We compared four different initialization strategies: using just one thread (serial), a `static` schedule with a `for` workshare construct, the `numactl` tool to enforce a round robin page distribution over NUMA nodes when the actual initialization is performed by one thread

Table 1. Average percentage of local and remote data accesses

	local iterations	remote iterations
`for` worksharing with `static` schedule	100%	0%
`for` worksharing with `dynamic` schedule	3.06%	96.94%
tasks-parallel single-producer	3.10%	96,90%
tasks-parallel parallel-producer	79.51%	20.49%

only, and finally tasks to initialize the data row-wise. Again, for the tasking variants we distinguish between the single- and the parallel-producer patterns. Fig. 2 shows that the `static` schedule, the round robin and the parallel-producer strategies result in a regular page distribution while for a serial initialization the complete memory is located on one NUMA node. However, the results also show that the single-producer pattern leads to an irregular distribution of pages, in which most are allocated on the 'single' node. This would lead to a serious performance degradation. The reason is that the initialization tasks are computationally very cheap and short-lived and the other threads on NUMA node 0 - besides the one performing the task creation - execute tasks at a fast enough pace. Task stealing does not occur in a noteworthy amount from other NUMA nodes.

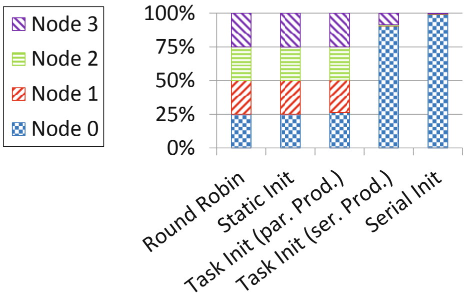


Fig. 2. Page distribution over the NUMA nodes after the matrix initialization

Multiple Levels of Parallelism. Composability of software components is not well-supported in OpenMP, i.e. worksharing constructs may not be nested within the dynamic extend of one single parallel region. Tasks can be nested inside other tasks and a worksharing construct as well, this particularly opens the opportunity for the parallel-producer pattern. The nesting of parallel regions has been supported by OpenMP early on, but only few application success stories have been reported [1]. Furthermore, nested parallel regions introduce several problems in general and on NUMA architectures in particular: (1) the thread teams for the inner parallel region are not guaranteed to be the same for two consecutive calls [5], thus data affinity cannot be maintained; and (2) the end of the inner parallel region always implies a barrier. These problems do not occur with nested tasks, especially with the techniques to create tasks we discussed so far. In Chap. 5 we show that for the FIRE code an implementation with nested tasks clearly outperforms an implementation with nested parallel regions.

Summing Up. We have shown that both task-parallel implementations perform the load balancing as well as OpenMP's `for` workshare with the `dynamic` schedule, but the parallel-producer pattern provides significantly better data locality. Tasks created in a thread bound to a particular NUMA node are picked up for execution on the same NUMA node, so that data locality is maintained if the same pattern is used during data initialization and the actual computation. This observation complies with the schedule strategies expressed in articles on OpenMP task implementations, as outlined in Chap. 2. Furthermore, one can expect the parallel-producer pattern to scale better than the single-producer in case of many small tasks, since the task creation occurs in parallel instead of being serialized. This will be analyzed in Chap. 4.

4 Task Behavior on NUMA Architectures

In this chapter we examine the behavior of two kernels, which both employ tasks, on a NUMA architecture. All measurements in this chapter have been performed on a bullx s6010 compute node, equipped with four Intel Xeon X7550 processors running at 2.0 GHz, offering 32 physical cores and 64 logical cores with hyper-threading, and 256 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.

4.1 STREAM

The first set of experiments has been carried out with the STREAM [9] benchmark. We picked this particular kernel to investigate effects of task-parallel implementations on NUMA architectures for two reasons: (1) if the data initialization is not done in the right way, the performance will be degraded significantly, as the computation performed in the individual kernels is memory-bound; and (2) the naive worksharing implementation delivers optimal performance if the same loop schedule is used during the data initialization and the actual computation. For the sake of brevity we only discuss results from the triad (daxpy) operation, since they are consistent with the other ones.

In Fig. 3 we compare the original parallel STREAM implementation referred to as *workshare: static-init for-loop* to several task-parallel versions. The original parallel version employs an OpenMP `for` worksharing construct with a `static` schedule 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. This results in an even data distribution over all NUMA nodes in the system. And as the computation is performed in the same manner, the number of remote accesses should be minimal. The arrays

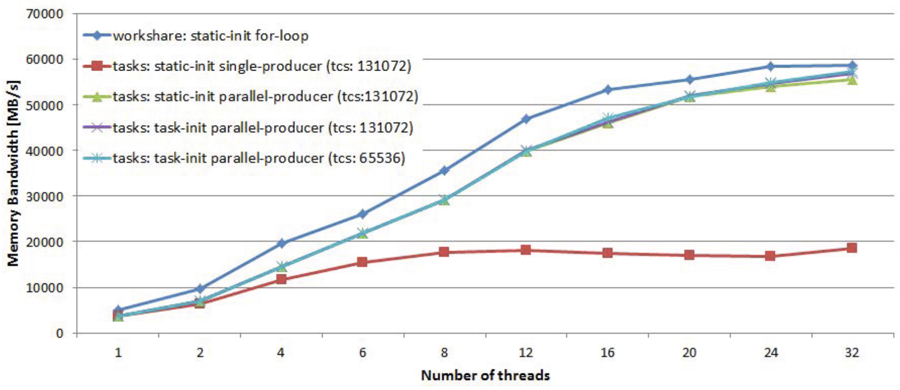


Fig. 3. STREAM triad operation: worksharing vs. task-parallel variants

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. Although the system offers 64 GB of memory per NUMA node, this kernel size is much larger than the cumulated cache size and thus we achieve reliable measurements of the memory bandwidth of the system. We implemented the following task-parallel variants, in all three the task chunk size (tcs) refers to the number of iterations grouped together in a single task:

- *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 within a `single` construct (*single-producer* 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).
- *tasks: task-init parallel-producer*: In this version both the data initialization and the computation is performed task-parallel by applying the same pattern to both code regions.

The results in Fig. 3 show that the worksharing version outperforms the best task-parallel version by just 3 %. The two task-parallel variants employing the parallel-producer pattern deliver approximately the same performance, as both distribute the data in an optimal fashion over the NUMA nodes. The single-producer tasking version clearly suffers from two effects: (1) the runtime cannot maintain data affinity, as all task 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. The parallel-producer pattern has to be used in the data initialization and the computation so that the OpenMP runtime is able to maintain data affinity. 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. As with the results discussed in the previous chapter, the task chunk size 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.

4.2 Sparse-Matrix-Vector-Multiplication in a CG-Method

While STREAM served our purpose as a simple benchmark indicating fine differences in the memory access pattern, the Sparse-Matrix-Vector-Multiplication (SMXV) in a CG-Method [8] 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 [6]. 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. As shown in Fig. 4 the sparsity pattern is slightly unbalanced regarding a `static` distribution.

Fig. 5 shows the SMXV performance when executing 1000 CG iterations. It compares the effect of different initialization strategies for both tasking patterns introduced in Chap. 3. The page distribution after the initialization of the sparse matrix correlates with the performance results of this experiment (see Fig. 2). As shown in Fig. 5(a), we reach a peak performance of 10 GFLOPS for the given workload. It also shows that using a `static` schedule for the data initialization is much better than using serial or a serial-producer task initialization. However, the performance for the `static` schedule decreases for more than 256 tasks while the random initialization still works well for 8192 tasks, which translates to a chunk size of 247 rows. It is obvious that the number of tasks is very important to reach the best performance. If too few tasks are used the load imbalance decreases the performance slightly. The overhead for the use of too many tasks dominates the runtime if chunks consists of only a few rows. Fig. 5(b) shows that for the parallel-producer pattern the peak performance reaches almost 13 GFLOPS. The performance of the round-robin initialization (10 GFLOPS) is comparable to the performance of the single-producer case, but the performance decline only occurs for

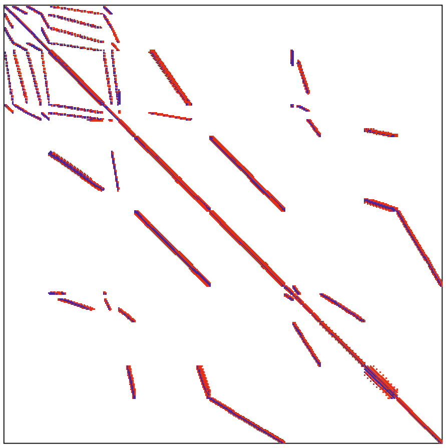
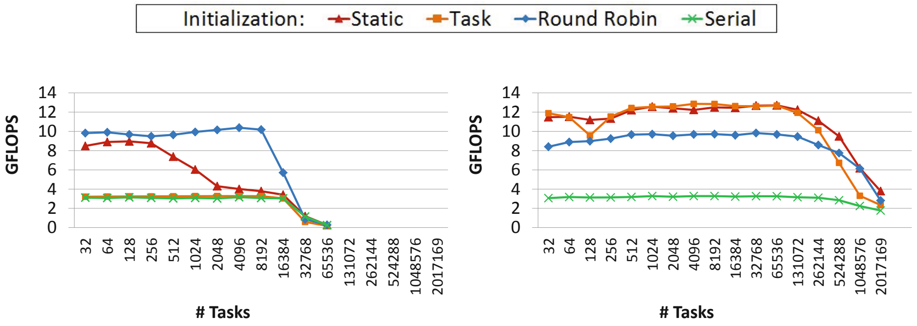


Fig. 4. Sparsity pattern of the matrix used in the CG-method



(a) Single-Producer Pattern (b) Parallel-Producer Pattern

Fig. 5. Performance of the SMXV kernel within the CG-method

more than 100,000 tasks and is not that significant. This proves that the task creation overhead in the OpenMP runtime is 'parallelized' in the parallel-producer pattern. The fact that results for the `static` schedule and the parallel-producer variants are better shows that the programmer has a much better influence on data locality by using this pattern.

5 Application Case Studies

In order to prove the applicability of the patterns and strategies we discussed so far, we employed them to two real-world application codes. In this chapter we show that our tasking implementation of TrajSearch reaches the same performane as the state of the art worksharing implementation, and for the FIRE code it event outperforms the corresponding worksharing variant. For all performance experiments in this chapter we use a Bull BCS system consisting of four bullx s6010 system as described in Chap. 4. The four systems are equipped with Bull's proprietary BCS cards providing a cache-coherent and high performant interconnect, creating a 128 core system with 16 NUMA nodes.

5.1 Trajectory Search

TrajSearch is a code to investigate turbulences which occur during combustion. It is a post-processing code for dissipation element analysis developed by Peters and Wang [12] from the Institute for Combustion Technology[1] at the RWTH Aachen University. It decomposes a highly resolved 3D turbulent flow field obtained by Direct Numerical Simulation (DNS) into non-arbitrary, space-filling and non-overlapping geometrical elements called 'dissipation elements'. Starting from every grid point in the direction of ascending and descending gradient of an underlaying diffusion controlled scalar field, the local maximum and minimum point are found. A dissipation element is defined as a volume from which all trajectories reach the same minimum and maximum point.

Every trajectory can be investigated independently from the others in parallel. A version of this code was parallelized with traditional worksharing and a different version has been parallelized with tasks. Fig. 6 (left) shows the performance results of tests done with both versions on the NUMA system. Due to the long execution time, we restricted our experiments to at least 16 threads. The `static` tests with the `for` worksharing construct perform slightly worse than the `dynamic` workshare and the task-parallel version. This is because the time for a single search for a trajectory is not constant, it depends on the length of the trajectory which is unknown a priori. This leads to some load imbalance and thus to a performance penalty when a `static` schedule is used. The `dynamic` parallel `for` loop and the tasking versions perform better, since the load is distributed among the threads more evenly. In conclusion it can be seen, that the load balancing capabilities of tasks for this application are as good as when a `for` worksharing loop with `dynamic` schedule is used.

5.2 FIRE

The Flexible Image Retrieval Engine (FIRE) [7] was developed at the Human Language Technology and Pattern Recognition Group[2] 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. The similarity is derived from comparing various image features. The existing parallelization of the FIRE code uses OpenMP [13] on two nested levels. On the outer level all query images are processed in parallel and on the inner level the comparison of one query image to the database images is also done in parallel.

We re-implemented the parallelization using OpenMP tasks. For every query image one task is created. Inside these tasks for every comparison of the query image to one

[1] `http://www.itv.rwth-aachen.de`

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

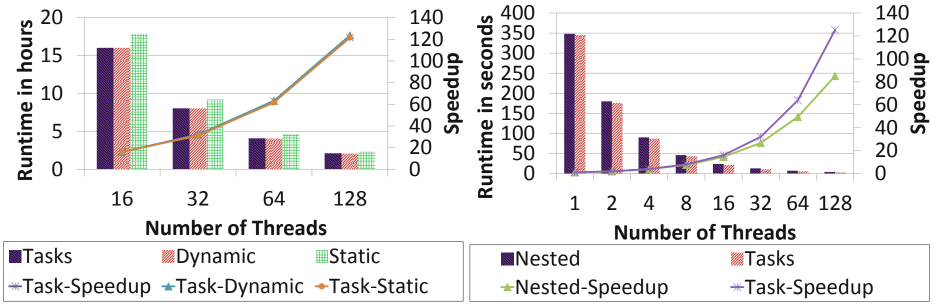


Fig. 6. Runtime and speedup of two application codes, TrajSearch (left) and FIRE (right). A comparison of tasking versions with a parallel for loop using different schedules for TrajSearch and with a version applying nested parallel regions for FIRE.

element of the database another task is created. Both parallel versions express the same amount of parallelism. Our test dataset comparing both versions processes 18 query images in a database consisting of 1000 images. The measured runtime and speedup on the 16-socket machine are shown in Fig. 6 (right). For the nested parallel regions the best combination of threads at the outer and inner regions has been used. E.g. the value for 16 threads is the minimum runtime for 1:16, 2:8, 4:4, 8:2 and 16:1 threads used at the outer:inner parallel regions.

Both versions of the code deliver nearly the same serial runtime, so the overhead of the OpenMP constructs is in the same order of magnitude. With more threads the tasking version outperforms the nested parallel region. For 128 threads the tasking version reaches a nearly linear speedup of 127 on 128 threads, whereas the nested parallel region only reaches a speedup of 85.

6 Conclusion

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 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. As we have shown, if thread binding is used and the task-parallelism is expressed using an appropriate pattern during both the data setup/initialization as well as during the actual computation, modern OpenMP runtimes, like the one from Intel we used, can maintain data affinity and thus achieve performance on par with state-of-the-art worksharing implementations.

Furthermore, with the real-world application use cases we have shown that tasking implementations may outperform the comparable worksharing implementations. This is particularly true for situations in which the load is not evenly balanced and a `dynamic` scheduling scheme is employed, in which case tasks may offer an even finer load balancing plus the ability to maintain data locality by applying the patterns presented above. This also extends to cases in which the worksharing approach is limited, such as when OpenMP parallel regions have to be nested.

References

1. an Mey, D., Sarholz, S., Terboven, C.: Nested Parallelization with OpenMP. International Journal of Parallel Programming 35, 459–476 (2007), 10.1007/s10766-007-0054-1
2. 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)
3. 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)
4. 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), doi:10.1007/s10766-010-0136-3
5. Terboven, C., an Mey, D., Schmidl, D., Jin, H., Wagner, M.: Data and Thread Affinity in OpenMP Programs. In: Proceedings of the 2008 Workshop on Memory Access on Future Processors: a Solved Problem?, MAW 2008, pp. 377–384. ACM (2008)
6. Davis, T.A.: University of Florida Sparse Matrix Collection. NA Digest 92 (1994)
7. Deselaers, T., Keysers, D., Ney, H.: Features for Image Retrieval - a quantitative comparison. Information Retrieval 11(2), 77–107 (2008)
8. 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)
9. McCalpin, J.: STREAM: Sustainable Memory Bandwidth in High Performance Computers
10. 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)
11. OpenMP ARB. OpenMP Application Program Interface, v. 3.1, `http://www.openmp.org`
12. Peters, N., Wang, L.: Dissipation element analysis of scalar fields in turbulence. C. R. Mechanique 334, 493–506 (2006)
13. Terboven, C., Deselaers, T., Bischof, C., Ney, H.: Shared-Memory Parallelization for Content-based Image Retrieval. In: ECCV 2006 Workshop on Computation Intensive Methods for Computer Vision (CIMCV), Graz, Austria (May 2006)
14. Terboven, C., Spiegel, A., an Mey, D., Gross, S., Reichelt, V.: Parallelization of the C++ Navier-Stokes Solver DROPS with OpenMP. In: Joubert, G.R., Nagel, W.E., Peters, F.J., Plata, O.G., Tirado, P., Zapata, E.L. (eds.) PARCO. John von Neumann Institute for Computing Series, vol. 33, pp. 431–438. Central Institute for Applied Mathematics, Jülich (2005)
15. 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)