Approaches for Task Affinity in OpenMP

Christian Terboven[1(✉)], Jonas Hahnfeld[1], Xavier Teruel[2], Sergi Mateo[2], Alejandro Duran[3], Michael Klemm[3], Stephen L. Olivier[4], and Bronis R. de Supinski[5]

[1] Chair for High Performance Computing, IT Center, RWTH Aachen University, Aachen, Germany
{terboven,hahnfeld}@itc.rwth-aachen.de
[2] Barcelona Supercomputing Center, Barcelona, Spain
{xavier.teruel,sergi.mateo}@bsc.es
[3] Intel, Santa Clara, USA
{alejandro.duran,michael.klemm}@intel.com
[4] Center for Computing Research, Sandia National Laboratories, Albuquerque, USA
slolivi@sandia.gov
[5] Lawrence Livermore National Laboratory (LLNL), Livermore, USA
bronis@llnl.gov

Abstract. OpenMP tasking supports parallelization of irregular algorithms. Recent OpenMP specifications extended tasking to increase functionality and to support optimizations, for instance with the `taskloop` construct. However, task scheduling remains opaque, which leads to inconsistent performance on NUMA architectures. We assess design issues for task affinity and explore several approaches to enable it. We evaluate these proposals with implementations in the Nanos++ and LLVM OpenMP runtimes that improve performance up to 40 % and significantly reduce execution time variation.

1 Introduction

The OpenMP* API specification first included support for task-based parallelism in version 3.0 [8]. In contrast to OpenMP worksharing constructs, task constructs support parallelization of irregular algorithms, e.g., code with recursion or graph traversals. The flexibility of OpenMP tasks leads to nondeterministic execution including highly dynamic mapping of tasks to threads.

Modern processor architectures do not provide uniform performance since an internal fabric connects multiple processor packages with their local memories to form a single shared memory system. This NUMA (non-uniform memory access) architecture exposes different memory latencies and bandwidth rates, depending on the memory location that is accessed. Two examples are the Intel® Quick Path Interconnect [13] of Intel® Xeon processors and the Bull Coherent Switch [2]. A typical strategy allocates data on its first touch to a physical page

The rights of this work are transferred to the extent transferable according to title 17 U.S.C. 105.

© Springer International Publishing Switzerland 2016
N. Maruyama et al. (Eds.): IWOMP 2016, LNCS 9903, pp. 102–115, 2016.
DOI: 10.1007/978-3-319-45550-1_8

in the local memory of the processor that issues the instruction. While OpenMP worksharing constructs can explicitly assign work to individual OpenMP threads, OpenMP tasks do not support this kind of control. Thus, tasking complicates control of page placement and memory locality.

In this paper, we assess issues that arise in extending OpenMP to support task affinity that would address this question. We propose two fundamental approaches to extend existing tasking constructs that provide hints to the OpenMP compiler and runtime that can guide the assignment of tasks to threads in order to improve data placement and memory locality. First, the novel `affinity` clause for the `task` construct suggests a place or thread on which to execute a task. Second, a `taskgroup` extension provides a hint to the runtime system about how to the distribute the tasks of that task group.

The remainder of the paper is organized as follows. Section 2 reviews prior proposals to support task affinity in OpenMP. Section 3 assesses key issues in the design of task affinity support while Sect. 4 presents the proposed syntax and semantics of our two approaches. Section 5 details the prototype implementations of our approaches while we provide a preliminary assessment of their performance in Sect. 6.

2 Related Work

Proposed OpenMP extensions for data, thread, and task affinity by Huang et al. predate offically-adopted OpenMP thread affinity support, but the article only evaluates data and thread, not task, affinity [5]. Terboven et al. found that the status quo of task scheduling – no mechanism to express affinity among tasks or between tasks and threads – leads to inconsistent performance across different OpenMP implementations and between different NUMA architectures [11]. Olivier et al. defined the concept of "work time inflation", additional time spent by threads in a multithreaded computation beyond the time required to perform the same work sequentially [7]. They showed the impact of such work time inflation in OpenMP task parallel programs and proposed OpenMP extensions to specify mappings of tasks to NUMA locality domains, enabling exploitation of first touch placement for tasks. Muddukrishna et al. base task scheduling on available capacity of the last level cache and working sets of the tasks [6]. Our work incorporates the preliminary lessons from many of these prior studies to provide a comprehensive assessment of the issues that arise for task affinity.

Work on task parallelism prior to the OpenMP task model has also examined the issue of task affinity. Acar et al. derive a theoretical bound on cache misses due to differences in the ordering of tasks between sequential and parallel executions [1]. They propose "locality-guided work stealing" that enqueues tasks with affinity to a particular thread into a special "mailbox" that is separate from its main queue. Task scheduling techniques based on system topology particular to other task-based programming models and runtime systems have also been attempted, e.g., for Charm++ [10] and Habanero [4,12]. Cilk's work-first task scheduler exploits task affinity naturally by design for programs in which

significant data sharing occurs between parent and child tasks [3]. Such data sharing often, but not always, occurs with recursive algorithms; it frequently does not for general task parallelism. Further, Cilk's design targets temporal locality in private caches, while modern systems have much more complex memory subsystem hierarchies: private and shared caches, multiple memory controllers and generally more complex memory systems. Our work extends OpenMP to assist OpenMP compilers and runtimes in using these increasingly complex memory systems.

3 Design Choices for Task Affinity

Substantially different issues arise with task affinity compared to thread affinity, which OpenMP 4.0 incorporated [9]. We explore several of these issues in this section.

How does task affinity limit task stealing? While OpenMP does not mandate a task scheduling policy, many implementations use task stealing to exploit OpenMP tasking semantics to improve load balance. Prescriptive task affinity extensions would prevent the implementation from exploring the trade-off between load balance and work time inflation. Thus, both of our approaches provide descriptive hints that can guide task scheduling decisions. While our proposals do not mandate task scheduling policies, the user may assume that hints bias task stealing to improve affinity.

How should affinity mechanisms interact with the task scheduling constraints? When a task encounters a task scheduling point it may switch (or not) to begin or resume the execution of a different task. These task switching points are subject to an explicit ruleset described in the OpenMP specification (i.e. the *Task Scheduling Constraints*). Thus, any task affinity scheduler implementation may use the information provided by the affinity clauses to guide task execution but they will always be subject to any constraint explicitly expressed in the OpenMP specification (see *descriptive* vs. *prescriptive* discussion in the previous paragraph).

Should tasks have affinity with threads or data? We could specify which OpenMP thread should execute a given task, which would support the distribution of tasks to threads to which the programmer has already carefully distributed data. Task-to-thread affinity also simplifies an initialization phase that must distribute data appropriately across system resources. Alternatively, we could specify data locations that are used by a given task to enable the task scheduler to execute each task on a resource that is close to its data. Task affinity to a data location may be the right level of abstraction for the programmer as it is independent of the underlying architecture or the data layout. However, the programmer must be able to specify all data important for affinity when the task is created (i.e. this data must be accessible at task creation). While straightforward for simple programs, data access sets of large programs with multiple compilation units are often not apparent where the tasks are created. Since both choices are useful in some cases, our proposals support both.

How should we express task-to-thread affinity? We could specify which thread should execute a task by using OpenMP places or by using OpenMP thread identifiers. Using OpenMP places would restrict the task to be executed by one of the threads bound to the given place(s). While this approach has conceptual appeal, the place list is static and defined through an environment variable before the program starts. Further, the place list heavily depends on the system architecture, which would require the programmer to have that architecture in mind when writing the program and, thus, fail to provide portable semantics. Because of this we decided not to support OpenMP places. Alternatively, while OpenMP thread identifiers may be appropriate if the program distributes data based on them, they also can limit portability and do not capture natural semantics for other data distribution strategies. Thus, our approaches provide this option but also can exploit another mechanism, which specifies higher level policies that capture task-to-thread affinity similarly to OpenMP thread affinity policies (i.e., spread or close). These policies allow the user to specify task-to-thread affinity independently of the exact number of threads or place list.

Which task-to-thread affinity policies should we support? Policies that are similar to those that already exist for thread affinity provide many advantages. First, many users are already familiar with the concepts expressed by those policies. Second, they have proven useful to guide efficient decisions for real applications. It has to be noted that a task affinity policy cannot directly be expressed in the context of the place list. To illustrate this, assume a parallel region with the `proc_bind(master)` clause, which determines all threads to execute within the same place. In this case, the `affinity(spread)` clause on a `taskgroup` construct cannot lead to task affinity outside of this single place, as task are executed by the threads in the current team. We also allow specification of task affinity at different levels of the task hierarchy, similarly thread affinity policies and nested parallelism.

Which memory accesses determine task-to-data affinity? Two types of memory accesses could guide task affinity: allocations or recent task (load/store) accesses. Task affinity could request that a task execute in the same place as the last task that used the same data, which would imply that the place to which task affinity refers changes when task stealing occurs. Alternatively, task affinity could refer to the place on which the data was first touched (allocated), which would fix affinity to the original place. The right choice is application dependent: compute-bound problems with good cache locality benefit from executing where the data was most recently accessed and likely is still resident while the data for memory-bound problems usually does not remain resident except where it was allocated. Thus, we support both patterns.

Should we use the depend clause to express task-to-data affinity? Since tasking constructs already support the `depend` clause to specify a relationship to data locations, we could use it to express task-to-data affinity since the same data location often captures data affinity and synchronization. However, the depend

clause tying affinity to synchronization semantics by binding the two concepts would violate the separation of concerns design principle. Therefore, we provide a new clause although we also support a short form that expresses the overlap when appropriate.

On which tasking construct should we express task affinity? Specifying task affinity on the `task` construct would make programs easier to read since the relationship is then visible where it applies. This choice easily supports task-to-data affinity. However, high-level task-to-thread affinity policies affect multiple tasks. Thus, specifying them on the `task` construct would be unclear and could lead to cases in which sibling tasks specify different policies. Thus, we also provide new clauses for thread affinity on the `taskgroup` construct, which clearly marks which tasks are affected by the policy.

It would be also desirable to have affinity support for tasks spawned from the `taskloop` construct. For task-to-thread affinity this is straightforward as spawned tasks are grouped by an implicit `taskgroup`. But to support task-to-data OpenMP currently lacks the language to be able to express the affinity of the different iterations and how that relates to spawned tasks. We therefore have decided to postpone a decision on how to handle task-to-data affinity for the `taskloop` construct.

4 Proposed Syntax and Semantics

As a general concept, this proposal introduces the `affinity` clause for the `task`, `taskgroup` and `taskloop` constructs, as discussed below.

Task: Task affinity could be expressed directly to an OpenMP place or thread, depending on the given specifier, as in the following example:

```
#pragma omp task affinity(thread: <thread-identifier>)
```

Task affinity could be also expressed by means of the data a task produces, modifies or consumes, indicated by a different specifier:

```
#pragma omp task affinity(data: A[i])
```

The task shall be executed as close as possible to the location of the specified data reference. Data location can be determined by the assumption of thread affinity (i.e. binding OpenMP threads to cores or sockets) and then grouping tasks that use the same location as close as possible.

A second approach could use system queries to determine where the data is actually allocated[1]. However, this option may be hard to implement: Since every task is executed by a thread, the runtime must first determine the physical location of the variable reference in the system and then perform a mapping into the place list to find the list of threads within that place, which are candidates to execute the task. On current Linux systems this incurs considerable overhead.

[1] Future versions of OpenMP may support explicit memory affinity and thereby inhance the definition of a location.

Taskgroup: The affinity to a set of other tasks cannot be expressed directly on a `task` construct, as it stands without the context of the other sibling tasks that may be generated during the execution of the program. In order to define a task distribution policy, the total number of tasks in the context must be known, as it would be with the `taskgroup` construct. Currently the `taskgroup` construct always includes an implied taskwait at the end, but in the following it is assumed that this could be omitted, for example with the introduction of a `nowait` clause.

The following example illustrates the use of the `affinity` clause on a `taskgroup` construct using the `spread` policy, analogous to the corresponding thread affinity policy:

```
#pragma omp taskgroup affinity(spread)
```

Task affinity cannot directly be expressed in the context of the place list, as explained above. To address this, we defined that task affinity `spread` such that the generated tasks shall be spread among the threads in the team, as far and evenly as possible. In the current implementation, the task distribution is determined based on the OpenMP thread ids – another option to implement the task distribution would be to consider the place list as well, which could be evaluated at a later point in time. Similarly, with the `close` policy, the generated tasks shall be executed closely together as far as appropriate in the context of the current thread team. And finally with the `master` policy, the generated tasks shall be executed by the master thread.

As will be discussed below, determining the set of tasks to be used for applying the policy may challenge the implementation, the definition of the policy could be extended with the specification of a number of tasks to be used together:

```
#pragma omp taskgroup affinity(spread:N)
```

In this scenario, the policy will be applied under the assumption that N tasks will be created in the construct. If more tasks are created the distribution will be restarted with task $N+1$ which may not deliver optimal performance as multiple tasks get scheduled on the same thread.

Taskloop: In its current form, the number of tasks to be generated is known. Consequently, the same task affinity policies discussed in the previous section are also useful in this case.

5 Prototype Implementations

As described above, some options were implemented and evaluated. Expression of task affinity with respect to a place or thread or storage location have been implemented in the Nanos++ runtime, while the task affinity policies on the `taskgroup` and `taskloop` constructs have been implemented in the LLVM runtime.

Nanos++: All changes to the runtime needed for the affinity support were limited to the scheduler submodule. The baseline is the Nanos++ distributed breadth first scheduler. This scheduler has a pair of ready queues per thread: local and private. The only difference in these two set of queues is that local queues allow stealing but private queues do not. Stealing can be enabled in all scheduler policies by means of an environment variable.

Private queues are only used once a tied task has been executed by a thread. At any task switching point, a tied task is queued in the private queue of the executing thread, preventing other threads from stealing the task.

Local queues are used when the task may still be executed by any thread. Usually the encountering task will enqueue newly created tasks in its own local queue. In this manner we ensure certain affinity guidance for multiple creator scenarios (e.g., task creation in loop worksharing constructs or nested tasks created in recursive programs). The only exception to this simple rule is when the runtime encounters a single threaded execution by an implicit task (i.e., inside a single or a master construct). In this case the scheduler policy distributes the work following a round-robin or random pattern (configurable by means of an environment variable) among all the threads of the team. There are cases in which the runtime fails and determines a single creation scheme when actually there are more threads creating tasks simultaneously (e.g., multiple single constructs with the nowait clause).

LLVM: We used the LLVM OpenMP runtime[2] to create a prototype implementation of task affinity policies for the `taskgroup` and `taskloop` constructs, as described in the subsection *Taskgroup* and *Taskloop*. In the LLVM runtime, OpenMP tasking is implemented with a local task queue for each thread. When a thread encounters a `task` construct, it creates a new task and puts it onto its local task queue. If a thread is idle and its local queue is empty, it will steal a task from another thread's queue.

5.1 OpenMP Place/Thread Approach

The extended Nanos++ runtime supports the *thread-id* mapping technique in a very straightforward way. The scheduler policy uses the set of threads local queues but it will target the corresponding queue using the thread identifier provided in the affinity clause.

5.2 Storage Location Approach

The implementation of the data-driven approach may imply different degrees of complexity. In the current discussion we will describe two different Nanos++ implementations when guiding the task affinity using data. In both cases thread local queues are still used in order to group these tasks with a certain affinity among them.

[2] http://openmp.llvm.org/.

The first (default) implementation determines the target thread-id using a hash-map function. All tasks providing the same memory address will be enqueued onto the same thread local queue. Our hash-map function relies on the pattern of consecutive memory blocks with the same size and we compute the thread-id by shifting to the right log2(size) the memory address and keeping the modulo number of threads.

The main problem with this approach is that information may not be accurate when stealing occurs. Once a task is enqueued in a thread local queue it should ultimately be executed by that thread. If stealing occurs then the task is actually executed by another thread but the rest of the tasks using the same data will still be scheduled to the formerly assigned thread.

A map can be used to keep track of the actual thread executing a task. The map can be updated when a task is stolen so that related tasks will also be scheduled on the thread that has stolen the task. This map is distributed among threads as mentioned above, so any thread will know in which map a given data can be found.

The latter implementation gets more accurate information for scheduling a task but has the associated cost of keeping track of where the tasks are distributed at different scheduling points: submission, dependence fulfilment and stealing. The evaluation section will give more information about the impact of this specific technique.

5.3 Taskgroup

In order to implement the task affinity policies, we followed the general approach to put each task onto the queue of the corresponding thread, which is determined according to the given policy. To evenly distribute the tasks of a `taskgroup` or `taskloop` construct over the available threads, in our approach it is required to know the total number of tasks to distribute.

Therefore with `affinity(spread)` all recently created tasks are collected in a dedicated list and only distributed among the threads when a `taskwait` is encountered, either explicitly or implicitly. This means that task execution has to be deferred until all tasks are created and ready to be distributed which may negatively impact the performance. With our proposal of `affinity(spread:N)`, the distribution and execution of tasks could start immediately with task creation as the thread to put the task on can be determined a-priori.

Each task maintains a dedicated list of threads to execute its child tasks. This list is partitioned according to the task affinity policy.

When using task affinity in a single producer pattern, and if the team consists of more threads than tasks created in a single recursion step or loop iteration, some threads will not immediately get a task to execute. These threads will try to steal tasks from other threads' task queues, which may disturb affinity. It is not desirable for that to occur too early. To ensure affinity is maintained until all threads are busy, we prevent any thread from stealing until it has at least executed one local task. This ensures that task stealing is still allowed, which is desirable as argued above to perform load balancing, for instance.

5.4 Taskloop

The LLVM OpenMP runtime recently gained support for taskloops, which internally makes use of taskgroups for synchronisation. Consequently we were able to reuse our implementation and extend the support for task affinity on the `taskloop` construct. It accepts the same task affinity policies as described in the semantics and implementation above.

6 Evaluation

Again, task affinity with respect to a place or thread or storage location on the one hand and the task affinity policies on the other hand have been implemented and evaluated differently, with the Nanos++ and LLVM runtimes, respectively (see Table 1 for an overview).

Table 1. Different approaches and their implementations.

Approach	Implementation	Evaluation
Place/Thread	Nanos++ (Subsect. 5.1)	Subsect. 6.1 (Fig. 1)
Storage Location	Nanos++ (Subsect. 5.2)	Subsect. 6.2 (Fig. 1)
Taskgroup Policy	LLVM (Subsect. 5.3)	Subsect. 6.3 (Fig. 2)
Taskloop Policy	LLVM (Subsect. 5.4)	Subsect. 6.4 (Fig. 3)

Measurements with the LLVM and Nanos++ runtimes have been performed on a two-socket Intel Xeon E5-2699 v4 (Broadwell) system, with 44 cores in total. This system exhibits a 2-level NUMA architecture with four memory domains, as the two sockets are each split into two rings and each ring is connected to its local memory controller.

Nanos++ runtime has also been evaluated in the MareNostrum III cluster. This system is based on Intel SandyBridge processors, iDataPlex Compute Racks, a Linux Operating System (based on a SuSe Distribution) and an Infiniband interconnection network. Each node has 2x Intel SandyBridge-EP E5-2670/1600 20M 8-core at 2.6 GHz and 8×4 GB DDR3-1600 DIMMS of memory.

In order to obtain the results presented in this section we used the STREAM synthetic benchmark[3]. The suite is composed by four different kernels described in Table 2 and each execution consists of multiple repetitions of these four kernels.

We evaluated our prototype implementation of task affinity for the `taskgroup` construct with a task parallel merge sort. This program is representative of the class of divide and conquer algorithms. The input size for the merge sort was 2^{33} integer values.

[3] Further information about the STREAM benchmark suite available at: http://www.cs.virginia.edu/stream/ref.html.

Table 2. The STREAM benchmark suite: description of kernels.

Name	Kernel	Bytes/Iteration	FLOPS/Iteration
COPY	a(i) = b(i)	16	0
SCALE	a(i) = q*b(i)	16	1
SUM	a(i) = b(i) + c(i)	24	1
TRIAD	a(i) = b(i) + q*c(i)	24	2

6.1 Place/Thread

We evaluated our place and thread approach described in Subsect. 5.1 using the aggregated results of the full STREAM suite. Figure 1 shows the performance results of executing this benchmark in SandyBridge and Broadwell respectively. Speedups are computed against the execution time of the very same parallel version without task affinity annotation and using a *per thread* (local queue) round-robin scheduler. The first two bars of each cluster of bars correspond to the thread approach.

SandyBridge results show that we have no penalty/no gain when running on a single socket, but we increase the performance up to 20 % when mapping tasks to threads when both sockets are used.

We have used different numbers of threads configurations when running on the Broadwell system: 11 threads (one ring of a single socket), 22 threads (all the cores of a single socket), 44 threads (all the cores of the two sockets) and 88 threads (enabling hyper-threading). As in the case of SandyBridge there is no penalty/no gain when running on a single NUMA domain but performance increases as we use a thread per core/two sockets (30 % of speed-up) and increases still more when using all threads on the two sockets (up to 44 %).

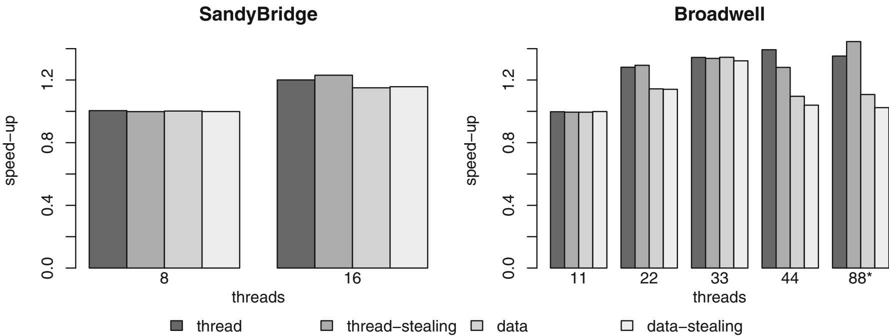

Fig. 1. Thread and storage (data) affinity approaches: STREAM benchmark performance with and without stealing, relative to a round-robin scheduler, on two different architectures using Nanos++.

The Broadwell results also show that stealing induces unpredictable execution behaviour. In some case we observe a performance degradation (e.g., with 44 threads) and in some other cases we the performance improves (e.g., with 88 threads). In this particular comparison (44 vs. 88 threads) the use of hyper-threading may also have an impact on the observed results due the plot also shows how the non-stealing version suffers a slight degradation in the speed-up gain.

6.2 Storage Location

Figure 1 also shows the results for the storage location approach, in the right two bars of each cluster of bars. In all cases we obtain a performance gain (i.e., the speedups are always bigger than 1) with respect to the non-affinity version, but comparing with the thread approach the performance gain is smaller. The gap between 22 and 44 threads in the Broadwell execution is an anomaly. In this specific case the storage affinity approach suffers from a small performance degradation while the thread approach is still able to improve results.

6.3 Taskgroup

The measurements discussed below were done with the Intel C/C++ Compiler in version 17.0 beta, employing our modified LLVM OpenMP runtime. 44 threads (one per physical core) were used, always delivering the best absolute performance.

Figure 2 shows that task affinity resulted in an improvement of approximately 20 % of execution time, and also a significant reduction of the performance variation between trials. Results for three versions of the program are shown. The *default* implementation performs data allocation and initialization in a sequential part, with the result that the whole array is located on only one NUMA domain. In the *first touch* variant, the array has been distributed over the NUMA nodes in chunks of equal size. The *affinity* variant employs the same data distribution together with a `omp taskgroup affinity(spread)` around the task creation points. Note that this currently still includes an implied taskwait synchronization construct at each recursion level, which we envision becoming optional in future versions of the OpenMP specification.

The improvement in execution time for the *affinity* variant stems from the higher percentage of local data accesses, as tasks are distributed according to the data distribution. The reduction in runtime variation occurs because the tasks' distribution to the threads based on the affinity policy is deterministic. In contrast, without task affinity the distribution is determined by stealing which in itself is nondeterministic. When stealing is allowed, data locality and therefore performance differs with every run.

6.4 Taskloop Construct

To evaluate affinity on the rather new `taskloop` construct, we modified the STREAM benchmark to use a single-producer pattern: the `taskloop` construct

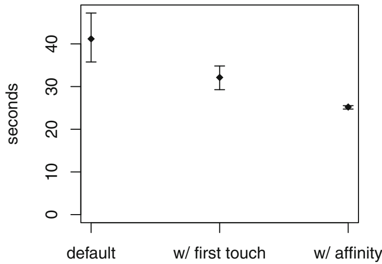

Fig. 2. Policy approach with taskgroup using LLVM: Avg, min and max execution time of merge sort with 2^{33} integer values.

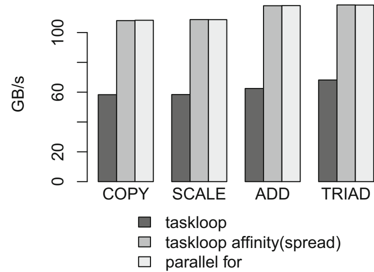

Fig. 3. Policy approach with taskloop using LLVM: STREAM benchmark performance.

is used to parallelize the loops performing the actual operations of the benchmark, and also the data initialization loops. On all instances the number of tasks to be created is set equal to the number of threads (`num_tasks(omp_get_num_threads())`): thereby the same number of explicit tasks is created as a `do` worksharing construct would create implicit tasks. As compiler support for this feature was very limited in the Intel compilers, we used a trunk version of the LLVM/Clang compiler. This version required us to use the `__builtin_nontemporal_store` intrinsic to enable the generation of non temporal stores. This was necessary to achieve maximum memory bandwidth with this code and to allow for a fair comparison. If task affinity is successful, this `taskloop` implementation should deliver the same memory bandwidth as the `do` worksharing variant.

Figure 3 compares three variants: the *taskloop* implementation as described with and without *affinity* enabled, and the same benchmark parallelized with the `parallel for` combined worksharing construct serving as the reference. In the version without affinity, the nondeterminism of the task to thread mapping in the task scheduling – as explained above – limits the achievable memory bandwidth. Enabling the `affinity(spread)` task affinity policy yields the same bandwidth as using the worksharing construct from the original STREAM benchmark – the ideal outcome.

7 Conclusion

In this paper we have discussed several language extensions to support task affinity in OpenMP. We focus on three different approaches. The first is based on the OpenMP places concept and complements OpenMP thread affinity. The second approach is based on data storage. This approach is more programmer-friendly (assuming programmers understand the data use of the tasks they create) but requires more complexity in the runtime implementation. The third approach is based on distribution policies of a set of tasks (e.g. those generated in `taskgroup` or `taskloop` constructs).

We have implemented and evaluated a representative prototype for each approach. The place-based approach is implemented assuming thread to core affinity. The storage approach uses the memory address as a key value to determine and group tasks using the same storage location. The distribution policies approach has been implemented by tracking at each task level the set of valid thread local queues a task can submit work to.

Results show that using this set of affinity guidelines when scheduling OpenMP tasks can help the runtime system to improve the application performance. Having different mechanisms to distribute tasks among threads or group their execution over the same (or a close) physical resource can help programmers to choose the one that fits best with their application. Regular and repetitive patterns of task creation may use thread-based task affinity, irregular patterns of memory usage may benefit from the ease of the storage-based approach, and recursive applications seem to fit with the task set distribution policies.

As future work we plan to further evaluate the different approaches on a wider set of kernels. We also plan to perform more in-depth experiments to better understand the effects of load imbalance and how stealing techniques may impact the performance. Finally, we plan to execute these kernels on additional system architectures to investigate the behavior of our implemented approaches in more complex system architectures (with respect to the NUMA layout). Our evaluations so far show significant benefits for OpenMP task parallel programs using the diverse approaches we investigated.

Acknowledgement. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

This work has been developed with the support of the grant SEV-2011-00067 of the Severo Ochoa Program, awarded by the Spanish Government, by the Spanish Ministry of Science and Innovation (TIN2015-65316-P, Computacion de Altas Prestaciones VII) and by the Intel-BSC Exascale Lab collaboration project.

Some of the experiments were performed with computing resources granted by JARA-HPC from RWTH Aachen University under project jara0001. Parts of this work were funded by the German Federal Ministry of Research and Education (BMBF) under grant numbers 01IH13008A(ELP).

Intel and Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands are the property of their respective owners.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations.

Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

References

1. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures, SPAA 2000, pp. 1–12. ACM (2000)
2. Bull Atos Technologies: Bull Coherent Switch. http://support.bull.com/ols/product/platforms/hw-extremcomp/hw-bullx-sup-node. Accessed 25 May 2016
3. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 1998, pp. 212–223. ACM (1998)
4. Guo, Y., Zhao, J., Cave, V., Sarkar, V.: SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010, pp. 341–342. ACM (2010)
5. Huang, L., Jin, H., Yi, L., Chapman, B.M.: Enabling locality-aware computations in OpenMP. Sci. Program. **18**(3–4), 169–181 (2010)
6. Muddukrishna, A., Jonsson, P.A., Brorsson, M.: Locality-aware task scheduling and data distribution for OpenMP programs on NUMA systems and manycore processors. Sci. Program. **2015**, 5:1–5:16 (2015)
7. Olivier, S.L., de Supinski, B.R., Schulz, M., Prins, J.F.: Characterizing and mitigating work time inflation in task parallel programs. In: Proceedings of the 24th International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 65:1–65:12. IEEE (2012)
8. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 3.0. http://www.openmp.org/
9. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 4.0. http://www.openmp.org/
10. Pilla, L.L., Ribeiro, C.P., Cordeiro, D., Bhatele, A., Navaux, P.O.A., Méhaut, J.F., Kalé, L.V.: Improving parallel system performance with a NUMA-aware load balancer. Technical reort TR-JLPC-11-02, INRIA-Illinois Joint Laboratory on Petascale Computing, Urbana, IL (2011). http://hdl.handle.net/2142/25911
11. 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)
12. Yan, Y., Zhao, J., Guo, Y., Sarkar, V.: Hierarchical place trees: a portable abstraction for task parallelism and data movement. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 172–187. Springer, Heidelberg (2010)
13. Ziakas, D., Baum, A., Maddox, R.A., Safranek, R.J.: Intel QuickPath interconnect architectural features supporting scalable system architectures. In: 2010 18th IEEE Symposium on High Performance Interconnects, pp. 1–6, August 2010