Description, Implementation and Evaluation of an Affinity Clause for Task Directives

Philippe Virouleau[1,2](✉), Adrien Roussel[1,2,3], François Broquedis[1], Thierry Gautier[1,2], Fabrice Rastello[1], and Jean-Marc Gratien[3]

[1] Inria, Univ. Grenoble Alpes, CNRS, Grenoble Institute of Technology, LIG, Grenoble, France
{philippe.virouleau,francois.broquedis, fabrice.rastello}@inria.fr, thierry.gautier@inrialpes.fr
[2] LIP, ENS de Lyon, Lyon, France
[3] IFPEN, Rueil Malmaison, France
adrien.roussel@inria.fr, jean-marc.gratien@ifpen.fr

Abstract. OpenMP 4.0 introduced dependent tasks, which give the programmer a way to express fine grain parallelism. Using appropriate OS support (such as NUMA libraries), the runtime can rely on the information in the *depend* clause to dynamically map the tasks to the architecture topology. Controlling data locality is one of the key factors to reach a high level of performance when targeting NUMA architectures. On this topic, OpenMP does not provide a lot of flexibility to the programmer yet, which lets the runtime decide where a task should be executed. In this paper, we present a class of applications which would benefit from having such a control and flexibility over tasks and data placement. We also propose our own interpretation of the new *affinity* clause for the *task* directive, which is being discussed by the OpenMP *Architecture Review Board*. This clause enables the programmer to give hints to the runtime about tasks placement during the program execution, which can be used to control the data mapping on the architecture. In our proposal, the programmer can express affinity between a task and the following resources: a thread, a NUMA node, and a data. We then present an implementation of this proposal in the Clang-3.8 compiler, and an implementation of the corresponding extensions in our OpenMP runtime libKOMP. Finally, we present a preliminary evaluation of this work running two task-based OpenMP kernels on a 192-core NUMA architecture, that shows noticeable improvements both in terms of performance and scalability.

Keywords: OpenMP · Task dependencies · Affinity · Runtime systems · NUMA

1 Introduction

OpenMP has become a major standard to program parallel applications on a wide variety of parallel platforms ranging from desktop notebooks to high-end

© Springer International Publishing Switzerland 2016
N. Maruyama et al. (Eds.): IWOMP 2016, LNCS 9903, pp. 61–73, 2016.
DOI: 10.1007/978-3-319-45550-1_5

supercomputers. It provides keywords to express fine grain task-based parallelism that boosts the applications performance and scalability on large-scale shared memory machines. In particular, tasking in OpenMP helps the programmers parallelize applications with an irregular workload, letting the runtime system be in charge of performing load balancing through task scheduling in a dynamic way. However, very little support exists to express and to control the affinity between tasks and data on systems with a decentralized memory layout, like *Non-Uniform Memory Architectures* (NUMA). On such systems, the memory is physically split into several banks, also called *NUMA nodes*, which leads to different memory latencies and throughputs depending on the location of the memory bank a core is accessing data from. To get the most performance out of such architectures, OpenMP runtime systems thus need to be extended to make the task scheduler aware of both the underlying hardware and the relation that exists between a task and the data it accesses.

We relate in this paper our experiences to reach high performance out of OpenMP numerical applications on a 192-core NUMA machine. The recently-added *places* concept in the OpenMP 4.0 specification provides ways of binding OpenMP parallel regions to user-defined partitions of the machine. This basically ends up binding the threads of the corresponding region to a set of cores. Thus, relying on the first-touch memory allocation policy as a portable solution to control memory binding, OpenMP places can help to control thread affinity with respect to the memory. However, the concept behind OpenMP places needs to be extended to improve the performance of task-based applications, as tasks are most of the time scheduled over threads in a dynamic way according to a work-stealing execution model. This is why the OpenMP *Architecture Review Board* is currently discussing the introduction of a new *affinity* feature to make the runtime system aware of the affinities between the tasks and the data they access.

In this paper, we present how we control task and data placement inside our OpenMP runtime system, implementing an *affinity* clause whose syntax is very close to the one currently discussed by the ARB. We also explain how we manage such information at runtime in order to improve the execution of task-based OpenMP programs on NUMA systems, with a particular focus on the scheduling data structure and the scheduling algorithm. The contribution of this paper is threefold:

- We propose an OpenMP *affinity* extension to the Clang-3.8 compiler able to express affinities between tasks and memory and pass this information along to the runtime system;
- We describe an extension to our task-based OpenMP runtime system to guide the scheduling of tasks according to such information to reach better performance on NUMA systems;
- We present some preliminary experimental results on running OpenMP benchmarks with tasks dependencies on a 192-core NUMA system, with and without using *affinity*.

The remainder of this paper is organized as follows. Section 2 introduces some motivating examples of applications that suffer from the lack of affinity support on NUMA machines. Section 3 details our proposal from the extension to the OpenMP specification to its actual implementation inside both the Clang compiler and our own OpenMP runtime system. Section 4 presents the performance evaluation of two OpenMP kernels that were enhanced to support affinity and were executed on a 192-core NUMA machine. We eventually present some related work in Sect. 5 before concluding in Sect. 6.

2 Motivating Examples for Which Affinity Does Matter

The high memory throughput of NUMA architectures has been introduced at the price of non-uniformity in memory latency. On such architectures, accessing local memory access induces lower latency than accessing data on a remote memory bank. To get the most performance, computational units of work, like *threads* and *tasks*, should ideally only access local memory.

Many projects from the High-Performance Computing research area deal with sparse linear solvers as fundamental building blocks. For instance, let us consider the BiCGStab [13] algorithm, a classical method for solving sparse linear algebra systems. Such algorithm is structured around a main loop that iterates until convergence is reached. At each iteration, the algorithm accesses global data through the computation of some sparse matrix-vector products as well as the execution of many global reductions like dot products. Preserving data locality among iterations is crucial to reach a high level of performance, especially for the sparse matrix products arising during the algorithm execution like reported by some early experiments running the BiCGStab algorithm (Sect. 4.3).

Another class of algorithms needing special care regarding data locality is the Stencil algorithms. These algorithms consist of multiple time steps during which every element of an array is updated using the value of its neighbors. Figure 1 shows the base performances of our Jacobi kernel, a stencil algorithm, evaluated on a 192-core NUMA architecture, with both Clang's OpenMP runtime and our OpenMP runtime libKOMP. We can see that the performances of either task-based versions are disappointing, as the execution time of this kernel increases when the number of threads is greater than 16. The reason behind this is that tasks are not scheduled close to their data. To do so, the runtime system should be aware of which data is accessed by every task and where the data has been physically allocated. While the former could be obtained through OpenMP data dependencies, the latter would need a specific support from the runtime level. Our proposal meets both these requirements through an OpenMP portable solution presented in the next section.

3 Extending OpenMP to Support Affinities

In this section, we detail our proposal with the introduction of the *affinity* keyword and how we implemented the corresponding runtime extensions that take advantage of this new feature.

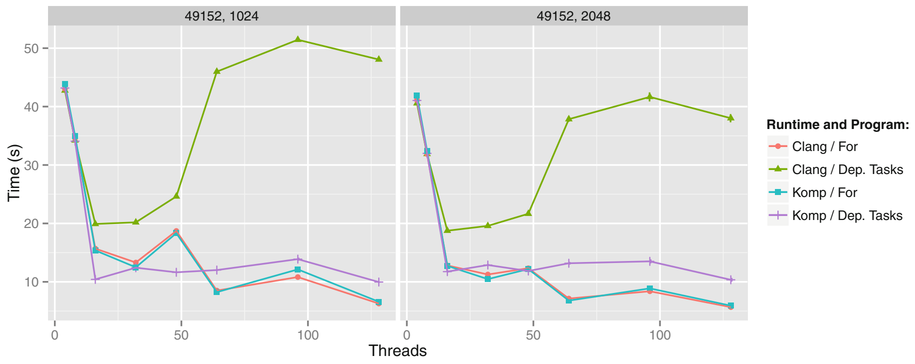


Fig. 1. Jacobi's base performances, with a Matrix size of 49152, and blocksizes of 1024 or 2048

3.1 Extension of the OpenMP Task Directive

We propose an extension to precisely control the *affinity* of a task with a specific part of the architecture hierarchy.

The two main components of NUMA architectures we consider in this work are cores and nodes. One of the key to getting performances out of NUMA architectures is to ensure tasks are executing *close* to their data. Therefore, we identified three different kinds of *affinity* the programmer may need to express, which are the following:

affinity to a thread: the runtime should try to schedule the task to be executed by a given thread.

affinity to a NUMA node: the runtime should try to schedule the task on any of the threads bound to a given NUMA node.

affinity to a data: once a task becomes ready for execution, the runtime should try to schedule it on any of the threads bound to the NUMA node on which the given data has been physically allocated.

Additionally, the programmer can specify if this affinity is *strict*, which means the task **must** be executed on the given resource, or not. In the latter case, the task scheduler may decide to execute the task on a different resource, to perform load balancing for example.

Since this extension is aimed for the tasking construct, we implemented it as a new clause for the OpenMP *task* directive. The proposed syntax for the clause is the following:

```
affinity([node | thread | data]: expr[, strict])
```

This proposal assumes the master thread with id 0 is executed on the first place in the place list. When *expr* refers to a thread id, it should refer to the thread id within the OMP_PLACES defined for the current team. For example, if the places for the current team are "{0},{1},{2}", thread with id 0 refers to "{0}". However, if the places are "{2},{5},{8}", thread with id 0 refers to "{2}".

When *expr* refers to a NUMA node id, it should refer to a node id within the set of NUMA nodes built from the OMP_PLACES list.

Two successive parallel regions with the same number of threads and the same places have the same set of NUMA nodes.

When *expr* refers to a data, it should be a memory address. If the NUMA node associated with the data can't be determined, it defaults to the first NUMA node of the team.

If *expr* refers to an out-of-bounds resource, the value is taken modulo the number of resources.

3.2 Extension of the OpenMP Runtime API Functions

In order to dynamically get information about the current team hierarchy, we also propose the following runtime API functions:

```
//Get the number of NUMA nodes in the team
omp_get_num_nodes(void);
//Get the NUMA node the task is currently executed on
omp_get_node_num(void);
//Get the NUMA node the data has been allocated on
omp_get_node_from_data(void *ptr);
```

These functions allow to query information about the hardware topology, and can only be called from inside a parallel region. On machines without NUMA support, we consider that all the threads are on a single NUMA node. In our proposed implementation, omp_get_node_from_data is implemented through Linux get_mempolicy interface.

We also added the following runtime API function that mimics the *affinity* clause:

```
//Set the affinity information to the next created
    tasks
omp_set_task_affinity(
      omp_affinitykind_t k, uintptr_t ptr, int strict);
```

The scope of the function call is the next created task in the current *task region*. This function takes an omp_affinitykind_t value (either omp_affinity_thread, omp_affinity_numa or omp_affinity_data) to specify which kind of affinity control is applied. value is either an integer that represents an identifier of the NUMA node, an identifier of a thread or an address in the process address space used to select the affinity NUMA node **when** the task becomes ready for execution.

We implemented these extensions in the Clang compiler, based on the 3.8 version[1]; and we also added the corresponding entry points in Clang's OpenMP runtime[2].

Please note only the entry points have been implemented in Clang's OpenMP runtime, the actual runtime support has only been implemented in our OpenMP runtime and is described in the following section.

3.3 Extension of the Task Scheduler to Support Affinity

We implemented extensions in the OpenMP runtime developed in our team, LIBKOMP [3,5], which is based on the XKAAPI [1,9] runtime system. XKAAPI is a task-based runtime system, using workstealing as a general scheduling strategy. This section gives a brief description of some of its key internal structures and mechanisms.

The Way XKAAPI Models the Architecture. XKAAPI sees the architecture topology as a hierarchy of `locality domains`. A `locality domain` is a list of tasks associated with a subset of the machine processing units. XKAAPI's locality domains are very similar to the notion of *shepherd* introduced in [11], or ForestGOMP's *runqueues* [2]. XKAAPI most of the time only considers two levels of domains : node-level domains, which are bound to the set of processors contained in a NUMA node, and processor-level domains, which are bound to a single processor of the platform. This way, at the processor level one `locality domain` is associated with each of the physical cores, and at the NUMA node level, one `locality domain` is associated with each of the NUMA nodes.

The Way XKAAPI Enables Ready Tasks and Steals Them. The scheduling framework in XKAAPI [1,9] relies on virtual functions for *selecting a victim* and *selecting a place* to push a ready task. When a processor becomes idle, the runtime system calls a function to browse the topology to find a locality domain, and steal a task from its task queue.

Implementation of the Support for Affinity. We extended the set of internal control variables (ICV) with an *affinity-var* property, and provided some runtime API functions to get and to set this ICV. As ICVs are inherited from the generating implicit task of the parallel region to each task this region generates, *affinity-var* can be considered as a per-task variable. The variable is composed of two fields: an `omp_affinitykind_t` value and an integer large enough to encode a pointer.

When a task construct using the *affinity* clause is encountered, the runtime sets the appropriate kind of affinity and the integer value in the ICVs. During task creation, these parameters will be set in the internal task descriptor.

[1] https://github.com/viroulep/clang.
[2] https://github.com/viroulep/openmp.

When a task becomes ready to be executed, the function responsible for the *selection of the place* to push the task will look at the affinity and select the appropriate locality domain. The capacity to defer the evaluation of the affinity until the task becomes ready allows the runtime to rely on the `get_mempolicy` function to identify the NUMA node on which a data is allocated.

As described earlier, an affinity can be *strict* or not. To implement this we used a private queue per locality domain. If the affinity is strict, the task is pushed to the locality domain's private queue. During the *victim selection*, a thread may only steal from the locality domain's public queue (in case of a locality domain attached to a NUMA node, every thread on this node can steal from the private queue).

4 Examples of Use and Experimentation Results

In this section, we describe two OpenMP kernels we extended to make use of the *affinity* clause. We also give some details on the platform we used to conduct experiments, before presenting the performance evaluation of different versions of these two kernels.

4.1 Enhancing Task-Based OpenMP Kernels to Support *affinity*

This section presents how we expressed affinities inside the two task-based OpenMP kernels we described in Sect. 2.

Jacobi. We looked into our Jacobi application from the KASTORS benchmark suite [14]. The application is a 2D stencil computational kernel that is repeatedly applied until convergence is detected. We used a blocked version of this algorithm. We used both a *dependent tasks* based implementation and a *for* based implementation. Each operation on a point of the matrix depends on its neighboring blocks, therefore the blocks should be physically evenly distributed among the nodes, and the computational tasks should be located close to these data.

Knowing the number of cores in the team, the matrix size and the block size, we computed a mapping between multiple neighboring blocks and the different cores.

We used the affinity clause to achieve two goals:

- first, to ensure the physical distribution of the data during initialization: in the dependent tasks version, each memory block is touched for the first time in the initialization task, therefore pinning the task to a thread ensures the memory will be physically allocated on its NUMA node. Listing 1.1 shows an example of the blocks initialization.
- second, to ensure tasks stay close to their dependencies during computation, by putting them on their block's thread.

We implemented both a strict affinity and a non-strict affinity version.

Listing 1.1. Example of use of the affinity clause for initialization

```
for (j = 0; j < ny; j+= block_size)
  for (i = 0; i < nx; i+= block_size) {
    #pragma omp task firstprivate(i,j) private(ii,jj)\
       affinity(thread:GET_PARTITION(i, j, block_size, nx, ny), 1)
    {
      for (jj=j; jj<j+block_size; ++jj)
        for (ii=i; ii<i+block_size; ++ii) {
          if (ii == 0 || ii == nx - 1 || jj == 0 || jj == ny - 1)
            (*unew)[ii][jj] = (*f)[ii][jj];
          else
            (*unew)[ii][jj] = 0.0;
        }
    }
  }
```

Sparse Matrix Vector Product. In this section, we present the *sparse matrix vector product* algorithms arising in the BiCGStab iterative algorithm. The main goal is to ensure that tasks will have local accesses to their data among the iterations. We split data following matrix graph partitioning techniques [13] while using automatic graph partitioner like Metis [7] tools.

In such a decomposition, a matrix A is split into several sub-domains of several rows: OpenMP independent tasks are responsible for computing sub-parts of the output vector. We ensure the task affinity using the common methodology in this paper: first data are allocated while taking care to evenly distribute them among the NUMA nodes while the workload is balanced among the cores; then we annotate tasks to constrain the scheduling.

To ensure an efficient data distribution on NUMA nodes, all the local data structures to a partition are allocated in parallel. Vectors are split following row permutations and splitting is dictated by partitions. Local parts of the vectors are distributed too (sparse matrix are stored in CSR format). Moreover, an output vector block associated with a part of the matrix is allocated on the same NUMA node than the partition itself.

The affinity of computational tasks are constrained by assigning them where partitions of the matrix are stored. This is very similar to the owner compute rule from HPF [8]: a task is mapped on the thread holding the output sub-vector $Y[i]$ (line 9 of Fig. 3(a)).

4.2 Experimental Platform Description

The machine we experimented on is an SGI UV2000 platform made of 24 NUMA nodes. Each NUMA node holds an 8-core Intel Xeon E5-4640 CPU for a total of 192 cores.

The memory topology is organized by pairs of NUMA nodes connected together through Intel QuickPath Interconnect. These pairs can communicate together through a proprietary fabric called NUMALink6 with up to two hops.

4.3 Experimental Results

Jacobi Kernel. We compared several blocked versions of the application with both the Clang's OpenMP runtime and the XKaapi runtime. The *jacobi_block_for* version uses *for* constructs during initialization and computation, while the *jacobi_block_taskdep* version generates tasks with dependencies for initialization and computation. Each version comes with or without using the *affinity* extension we propose. We refer to these enhanced versions as *jacobi_block_for_affinity* and *jacobi_block_taskdep_affinity*. The last enhanced version is the *jacobi_block_taskdep_affinity_nonstrict*, which uses a strict initialization, but a non-strict affinity for tasks during computation.

The initialization part of the *jacobi_block_for_affinity* uses tasks instead of the regular *for* construct, so that we could use the affinity clause and precisely set which thread initialize which data. The computation part of the algorithm has not been changed, there is no affinity during the computation.

Matrix size and block sizes have been chosen so that partitioning easily match the number of threads up to 128. Experiments have been made with a block size of 1024 or 2048, and with a matrix size of 49152.

Base performances comparison between Clang's runtime and XKaapi are available on Fig. 1 from Sect. 2.

Figure 2 focuses on results for XKaapi used through libKOMP.

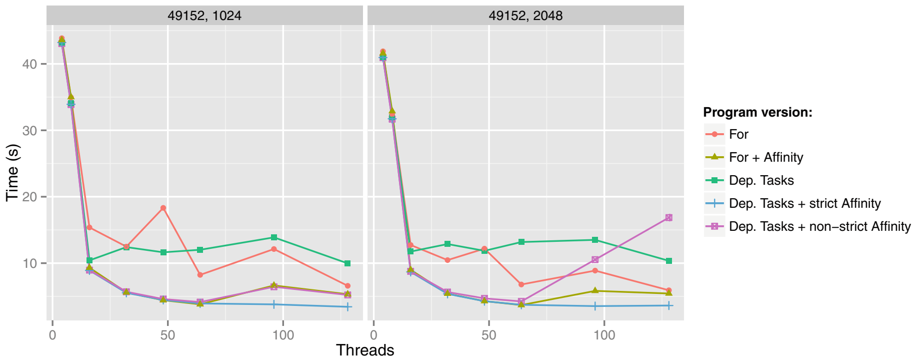


Fig. 2. Jacobi's performances overview using libKOMP, with a Matrix size of 49152, and blocksizes of 1024 or 2048

A general comment on these results is that the application globally does not scale well, whichever runtime or version is used. The program is memory bound and there is not much we can do besides ensuring computation occurs close to the data, in order to minimize the impact of memory bandwidth. In all these results, only the use of the *affinity* extension prevent a severe decrease in performances when increasing the number of threads.

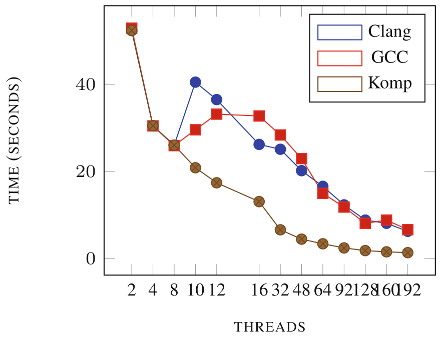
The basic dependent *tasks* version offers really poor performances, the basic *for* version is a bit better but still has room for improvement. The two high results for the *for* versions in Figs. 1 and 2 are obtained for a number of threads of 48 and 96: these numbers are not powers of 2 (whereas all the other number of threads are), and are not automatically perfectly mapped on the topology. For these numbers the mapping of the blocks on the architecture is not a perfect square, therefore each thread needs data from more neighbors, and a slight shift in initial iterations placement leads to worse performances.

Interestingly, using a strict affinity during initialization is beneficial for both *for* and *task* version: we can ensure a balanced mapping of the data over the whole hierarchy, even with non-square numbers.

As described in Sect. 2, the Jacobi kernel is a stencil algorithm and is very sensible to data locality and cache reuse. It explains why the version using dependent tasks with strict affinity achieves better performances than the non-strict version, where tasks may be stolen from a remote node, therefore ruining the cache reusability and the data locality (this is especially true with bigger blocks).

```
Compute Y = A × X
A sparse matrix
X, Y vectors
/* omp parallel region outside the
      function */
for (i=0; i < n_partitions; ++i)
{
        #pragma omp task depend(in: X[])
             depend(out: Y[i]) \
                 affinity(data: Y[i], 1)
        csr_mult( A.part[i], X, Y[i]);
}
```

(a) OpenMP SpMV Algorithm



(b) Results on our 192-core NUMA machine.

Fig. 3. SpMV experiment

Sparse Matrix Vector Product (SpMV Operation). In our experiment, 500 iterations of SpMV operations are timed and the average times is reported in Fig. 3(b) using various number of cores p. The matrix here corresponds to a Finite Volume discretization of a 2D Laplace problem on a square mesh of size 2000×2000. We run the same code compiled and executed with Clang-3.8 and its standard OpenMP runtime (labeled `Clang` on Fig. 3), GCC-5.2 with libGOMP (`GCC`) and our modified Clang-3.8 compiler with our OpenMP runtime libKOMP (`Komp`).

Up to 8 cores, all the execution times decrease in the same way for the three configurations `Clang`, `GCC` and `Komp`: differences between them is not visible. When the number of cores exceeds 8, `Clang` and `GCC` have execution times that increase before to decrease with a growing number of cores. On our machine, a NUMA node is composed of 8 cores. When $p > 8$ then the program has to use several NUMA nodes. For both `Clang` and `GCC` this is due to the misplacement of tasks on NUMA nodes where accessed memory is allocated. Data are split to fit on the local memory of each NUMA node. Vector are split into several parts, which are allocated on different NUMA nodes by using initialization tasks placement, relying on the OS first-touch policy. Matrices are split and also allocated by the use of initialization tasks so that it matches the same NUMA nodes on which the corresponding sub-vector has been allocated. Despite this tasks misplacement, the computation times are still decreasing because computations related to a domain are always done on the same core due to scheduling policies offered by `Clang` and `GCC`. XKaapi obtains better results because of the use of affinity clauses to place tasks on specific NUMA nodes, which ensures the temporal affinity among the iterations.

5 Related Work

Many research projects have been carried out to improve the execution of OpenMP applications on NUMA machines.

The HPCTools group at the University of Houston has been working in this area for a long time, proposing compile-time techniques that can help improving memory affinity on hierarchical architectures like distributed shared memory platforms [10]. Huang et al. [6] proposed OpenMP extensions to deal with memory affinity on NUMA machines, like ways of explicitly aligning tasks and data inside logical partitions of the architecture called *locations*.

Drebes et al. [4] proposed scheduling techniques to control both the data placement and the task placement, in order to take advantage of the data locality. They implemented these techniques in dataflow programming model named OpenStream. Their approach is focused at a scheduler level and does not provide flexibility to the user regarding data placement.

Olivier et al. [12] introduced node-level queues of OpenMP tasks, called *locality domains*, to ensure tasks and data locality on NUMA systems. The runtime system does not maintain affinity information between tasks and data during execution. Data placement is implicitly obtained considering that the tasks access memory pages that were allocated using the *first-touch* allocation policy. The authors thus ensure locality by always scheduling a task on the same locality domain, preventing application programmers to experiment with other memory bindings.

The INRIA Runtime group at the University of Bordeaux proposed the ForestGOMP runtime system [2] that comes with an API to express affinities between OpenMP parallel regions and dynamically allocated data. ForestGOMP implements load balancing of nested OpenMP parallel regions by moving branches of the corresponding tree of user-level threads on a hierarchical way. Memory affinity information is gathered at runtime and can be taken into account when performing load balancing.

6 Conclusion

OpenMP 4.0 introduced dependent tasks, which give the programmer a way to express fine grain parallelism that can be dynamically mapped to the architecture topology at runtime. Controlling data locality is one of the keys to performance when targeting NUMA architectures, and on this topic, OpenMP does not provide a lot of flexibility to the programmer yet, which leaves the responsibility to the runtime to make choices regarding tasks placements.

In this paper, we presented a class of applications which would benefit from having such a control and flexibility over tasks and data placement.

We proposed an implementation of a new *affinity* clause for the *task* directive, based on the discussion within the OpenMP language committees. It enables the programmer to give hints to the runtime about tasks placement during the program execution. These hints, combined with NUMA's first touch policy for memory, can be used to control the data mapping. The programmer can express affinity between a task and the following resources: a thread, a NUMA node, and a data.

We implemented this proposal in the Clang-3.8 compiler, and implemented the corresponding extensions in our OpenMP runtime libKOMP.

Finally, we performed a preliminary evaluation of this work running two task-based OpenMP kernels on a 192-core NUMA architecture, that showed noticeable improvements both in terms on performance and scalability.

In future, our focus will move to compile-time techniques able to infer and attach valuable information to tasks, like an estimation of a task operational intensity, that could guide some of the runtime system's decisions regarding task scheduling, load balancing, and data placement. We strongly believe a tight cooperation between the compiler and the runtime system is a key step to enhance the performance and scalability of task-based programs on large-scale platforms.

Acknowledgments. This work is integrated and supported by the ELCI project, a French FSN ("Fond pour la Société Numérique") project that associates academic and industrial partners to design and provide a software environment for very high performance computing.

References

1. Bleuse, R., Gautier, T., Lima, J.V.F., Mounié, G., Trystram, D.: Scheduling data flow program in XKaapi: a new affinity based algorithm for heterogeneous architectures. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014 Parallel Processing. LNCS, vol. 8632, pp. 560–571. Springer, Heidelberg (2014)
2. Broquedis, F., Furmento, N., Goglin, B., Wacrenier, P.-A., Namyst, R.: ForestGOMP: an efficient OpenMP environment for NUMA architectures. Int. J. Parallel Programm. **38**(5), 418–439 (2010)
3. Broquedis, F., Gautier, T., Danjean, V.: libKOMP, an efficient OpenMP runtime system for both fork-join and data flow paradigms. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 102–115. Springer, Heidelberg (2012)

4. Drebes, A., Heydemann, K., Drach, N., Pop, A., Cohen, A.: Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. ACM Trans. Archit. Code Optim. **11**(3), 30:1–30:25 (2014). Special Issue on OpenMP; Müller, M.S., Ayguade, E. (eds.)
5. Durand, M., Broquedis, F., Gautier, T., Raffin, B.: OpenMP in the Era of Low Power Devices and Accelerators, pp. 141–155. Springer, Berlin, Heidelberg (2013)
6. Huang, L., Jin, H., Yi, L., Chapman, B.: Enabling locality-aware computations in openmp. Sci. Program. **18**(3–4), 169–181 (2010)
7. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
8. Kennedy, K., Koelbel, C., Zima, H.: The rise and fall of high performance fortran: an historical object lesson. In: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, HOPL III, pp. 7-1–7-22. ACM, New York (2007)
9. Lima, J.V.F., Gautier, T., Danjean, V., Raffin, B., Maillard, N.: Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures. Parallel Comput. **44**, 37–52 (2015)
10. Marowka, A., Liu, Z., Chapman, B.: Openmp-oriented applications for distributed shared memory architectures: research articles. Concurr. Comput. Pract. Exper. **16**, 371–384 (2004)
11. Olivier, S., Porterfield, A., Wheeler, K.B., Spiegel, M., Prins, J.F.: Openmp task scheduling strategies for multicore NUMA systems. IJHPCA **26**(2), 110–124 (2012)
12. 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 International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012 (2012)
13. Saad, Y.: Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, 2nd edn. SIAM, Philadelphia (2003)
14. Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., Gautier, T.: Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In: DeRose, L., Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 16–29. Springer, Heidelberg (2014)