Using data dependencies to improve task-based scheduling strategies on NUMA architectures

Philippe Virouleau, François Broquedis, Thierry Gautier, Fabrice Rastello

Using data dependencies to improve task-based scheduling strategies on NUMA architectures

Philippe Virouleau, François Broquedis, Thierry Gautier, and Fabrice Rastello

Inria, Univ. Grenoble Alpes, CNRS, Grenoble Institute of Technology, LIG, Grenoble, France
LIP, ENS de Lyon, France
firstname.lastname@inria.fr
thierry.gautier@inrialpes.fr

Abstract. The recent addition of data dependencies to the OpenMP 4.0 standard provides the application programmer with a more flexible way of synchronizing tasks. Using such an approach allows both the compiler and the runtime system to know exactly which data are read or written by a given task, and how these data will be used through the program lifetime. Data placement and task scheduling strategies have a significant impact on performances when considering NUMA architectures. While numerous papers focus on these topics, none of them has made extensive use of the information available through dependencies. One can use this information to modify the behavior of the application at several levels : during initialization to control data placement and during the application execution to dynamically control both the task placement and the tasks stealing strategy, depending on the topology. This paper introduces several heuristics for these strategies and their implementations in our OpenMP runtime XKAAPI. We also evaluate their performances on linear algebra applications executed on a 192-core NUMA machine, reporting noticeable performance improvement when considering both the architecture topology and the tasks data dependencies. We finally compare them to strategies presented previously by related works.

Keywords: *OpenMP, task dependencies, benchmark, runtime systems, NUMA,* XKAAPI*, scheduling, work-stealing*

1 Introduction

While non-uniform memory access (NUMA) architectures stand today as one of the most popular design to build large-scale shared memory machines, exploiting them at their full potential remains challenging. On such architectures, the memory is split into several NUMA nodes and both bandwidth and latency depend on which processor accesses specific data : accessing memory allocated locally is most of the time faster than accessing data allocated to remotely-located NUMA nodes. Controlling data locality over the application lifetime is one of the key steps to achieving both good performance and scalability on these architectures.

Task-based parallel programming environments like OpenMP have become very popular when it comes to program shared memory machines with hundreds of cores. Indeed, they offer ways of expressing massive fine-grain parallelism with a relatively low overhead. Most of them also come with facilities to dynamically perform load balancing of tasks over the processors. Even if such characteristics fill the need of generating

more and more parallelism out of parallel applications, standard parallel programming environments still not explicitly address the problem of data locality on NUMA systems.

The runtime system plays a central role in the execution of a task-based parallel application. For example, it is responsible for assigning ready tasks to the target platforms' processors. It is also in charge of performing load balancing when a processor idles. Both these decisions should take the architecture topology into account in order to avoid NUMA-related performance penalties on the overall application performance.

The recent addition of data dependencies to the OpenMP tasking model provides the runtime system with very precise information about which part of an application accesses which variables. Thanks to these dependencies, the runtime system knows which memory areas are read or written by which task. As shown in this paper, the task scheduler can rely on this information when assigning tasks to processors to implement NUMA-aware strategies.

This paper describes several of these strategies we implemented inside the XKAAPI [9] runtime system. We identified three major steps in the task scheduler workflow that may have an impact on parallel applications on NUMA systems : the data distribution, the assignment of ready tasks to the processors and the way the task scheduler browses the architecture topology to perform load balancing. This paper describes and evaluates them, showing how they impact the application performance on a 192-core NUMA machine. We also compare them to state-of-the-art task scheduling strategies taken from related works and implemented within XKAAPI.

The layout of this paper falls into six sections as follows. In Section 2, we first give some background on NUMA architectures and the task programming model with data dependencies. We then describe in Section 3 the ideas, strategies and implementation details that we used to improve the runtime performances for these applications. Section 4 is devoted to the presentation performances evaluation. We eventually present some related works in Section 5 before concluding.

2 NUMA architectures design and exploitation

2.1 Hardware background

Most of nowadays parallel shared memory architectures are built according to a NUMA design where the memory is physically split into several banks attached to processors. Many vendors assemble these banks in a hierarchical way, thus building shared memory machines embedding several hundreds of cores. Exploiting such architectures at their full potential requires a fine control of the execution of a parallel application, as accessing local memory is most of the time faster than accessing memory stored in a memory bank attached to a remote processor.

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. We refer to this machine as Intel192 in the paper. 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.

Table 1 shows the distances advertised by the hwloc library [4] that represents the communication time for different distances normalized to the time of a local communication. Distances named *local* and *peer* form a pair of NUMA nodes (through Intel QPI), other nodes are either one hop away or two hops away (through NUMALink6).

Table 1: NUMA distances from node 0 advertised by the hwloc library on Intel192.

NUMA nodes location	*local*	*peer*	*one hop away*	*two hops away*
hwloc distances	1.0	5.0	6.5	7.9

2.2 Software background

To exploit large-scale shared memory architectures, the application programmer needs:

1. to express massive fine grain parallelism to get the most out of the numerous processing units of the platform ;
2. to control the execution of the application, especially the way computations and data are distributed over the platform, to prevent the NUMA design to have a negative impact on the overall application performance.

Task-based parallel programming environments provide ways of expressing fine grain parallelism that can be dynamically assigned to processors at runtime. OpenMP [12], the de-facto standard for shared-memory parallel programming, supports task parallelism with dependencies since revision 4.0.

A glimpse at OpenMP tasking An OpenMP *task* can be seen as an independent *unit of work* an OpenMP thread can execute. Tasks can be created by an OpenMP thread and executed by any thread of the same parallel region. As managing tasks at runtime is way cheaper than creating and synchronizing threads, the application programmer can take the parallelization of its application further, as he can now consider portions of code that were too fine grain to be parallelized using only threads. The synchronization of OpenMP 3.0 tasks is performed thanks to the `taskwait` keyword that waits for the completion of all the tasks generated from the current OpenMP parallel region. On one hand, the application programmer is responsible for creating and synchronizing OpenMP tasks explicitly. On the other hand, the runtime system is in charge of correctly assigning tasks to threads during the application execution.

OpenMP 4.0 pushes the concept of task further introducing the `depend` keyword to specify the access mode of each shared variable a task will access during its execution. Access modes can be set to either `in`, `out` or `inout` whether the corresponding variable is respectively read as input, written as output or both read and written by the

```
1  for (size_t k=0; k < NB; ++k) {
2  #pragma omp task shared(A) \
3    depend(inout: A[k][k])
4    dpotrf(NB,&A[k][k]);
5
6    for (int m=k; m < NB; ++m)
7  #pragma omp task shared(A)\
8    depend(in: A[k][k]) \
9    depend(inout: A[m][k])
10     dtrsm(NB,&A[k][k],&A[m][k]);
11
12   for (int m=k; m < NB; ++m) {
13 #pragma omp task shared(A)\
14   depend(in: A[m][k]) \
15   depend(inout: A[m][m])
16    dsyrk(NB,&A[m][k], &A[m][m]);
17
18     for (int n=k; n < m; ++n)
19 #pragma omp task shared(A)\
20   depend(in: A[m][k],A[n][k])\
21   depend(inout: A[m][n])
22      dgemm(NB,
23        &A[m][k],&A[n][k],&A[m][n]);
24   }
25  }
```

Fig. 1: Cholesky factorization with OpenMP-4.0 task dependencies

considered task. This information is then processed by the underlying runtime system to decide whether a task is ready for execution or should first wait for the completion of other ones.

KASTORS Benchmark suite Listing 1 shows the implementation of a Cholesky factorization implemented with OpenMP task dependencies. This factorization algorithm comes from the PLASMA library and is very similar to the one implemented in the KASTORS benchmark suite [15]. Task dependencies support comes with several benefits. First, task dependencies involve decentralized, selective synchronization operations that should scale better than the broad-range taskwait-based approaches. In some situations, this way of programming unlocks more valid execution scenarios than explicitly synchronized tasks, which provides the runtime system with many more valid task schedules to choose from. For example, in the Cholesky factorization, many instances of the `dtrsm`, `dsyrk` and `dgem` BLAS computations can legally run concurrently when executing the version with task dependencies. Secondly, information about task dependencies also enables the runtime system to optimize further, such as improving tasks and data placement.

The way we execute task-based applications Most task-based programming environments rely on a work-stealing execution model, originally introduced in Cilk [8]. Work-stealing is indeed often considered when it comes to dynamically balance the workload among processing units. The work-stealing principle can be summarized as follows. An idle thread, called a thief, initiates a steal request to a randomly selected victim. On reply, the thief receives a copy of one ready task, leaving the original task marked as stolen. Coherency between a thief and its victim is ensured by a variant of Cilk's T.H.E protocol, also described in [8].

The runtime system we develop, called XKAAPI, also implements the work-stealing execution model to execute OpenMP task-based applications. The runtime creates a system thread, called a *kproc*, for each processing unit to be used. On a NUMA multicore machine, a processing unit is a core. A kproc creates tasks and pushes them on its own work queue, which is implemented as a stack. The enqueue operation is very

fast : it takes around ten cycles on modern x86/64 processors [3]. As in Cilk, a running XKAAPI task can create children tasks. Depending on the number of tasks per thread, XKAAPI implements two strategies to find a ready task. If the number of tasks is lower than a threshold, XKAAPI follows the Cilk's *work first principle*. In that case, the thief iterates through the victim's stack queue from the least recently pushed task to the most recently one and it computes true data-flow dependencies for each task until a ready task is found. If the number of tasks is greater than the threshold, the data flow graph is built and the thief picks a task from the victim's list of ready tasks [1]. By the nature of our benchmarks, this latter strategy is de facto selected and we developed new NUMA aware strategies.

3 Using OpenMP tasks dependencies to improve tasks and data placement on NUMA machines

In this section, we describe how the runtime system can have a positive impact on the application execution using the information provided by data dependencies. The following sections describe the way we adapted the behavior of the runtime system to control data placement during the initialization phase, when data will be allocated and accessed for the first time, and how we modified the way tasks that perform the actual computations are dynamically assigned to processors while maximizing data locality.

3.1 Inside the XKAAPI task-based runtime system

This section describes some of the key internal structures and mechanisms of the XKAAPI runtime system.

The way XKAAPI models the architecture. XKAAPI sees the architecture topology as a hierarchy of `places`. A `place` is a list of tasks associated with a subset of the machine processing units. XKAAPI's places are very similar to the notion of *shepherd* introduced in [10], or ForestGOMP's *runqueues* [2]. XKAAPI most of the time only considers two levels of places : node-level places, which are bound to the set of processors contained in a NUMA node, and processor-level places, which are bound to a single processor of the platform. This way, at the processor level one `place` is associated to each of the physical cores, and at the NUMA node level one `place` is associated to each of the NUMA nodes.

The way XKAAPI enables ready tasks and steals them. The scheduling framework in XKAAPI [1] relies on virtual functions for *selecting a victim*, *selecting a place* to push a ready task and *pushing* a set of initial ready tasks.

When a processor becomes idle the runtime system has to *select a victim* and calls a function, called `WSselect` for *work-stealing select*, to browse the topology to find a place from which stealing a task from the place task queue.

The completion of a task may unlock the execution of some of its children in the dependency graph. This means marking them as ready for execution and pushing at least

one of them to a place. Once again, there are many ways of selecting the place where to push ready tasks, implemented in strategies we refer to as `WSpush`, for *work-stealing push*.

Before parallel computation begins, the runtime system can distribute (*push*) the set of ready tasks to multiple places, according to the strategy defined by `WSpush_init`.

These **three functions** are the main entry points to specify a **scheduling algorithm** in XKAAPI. Sections 3.3 to 3.5 describe strategies for these three points, all them were designed to explore the possibilities of the target NUMA architectures, to be able to evaluate which one are worth taking into account.

3.2 Controlling data distribution on a NUMA system

Controlling the way data are allocated on a NUMA system requires a good understanding of the underlying memory architecture. Application programmers can achieve this using dedicated tools or libraries, like libNUMA's `numactl` [17], which can be used to set a default memory allocation policy for the whole application. For example, the `--interleave=all` memory policy spreads out all the memory pages of dynamically allocated variables, over all the NUMA nodes of the machine. This policy is widely used on NUMA systems in conjunction with dynamic parallelism, like task-based programs, as it distributes the memory traffic over all the memory controllers, making processors "*all equally bad*" when it comes to memory access. To better control data placement, parallel application programmers are used to relying on the *first-touch* allocation policy, which is the default behavior for memory allocation on most Linux systems. This allows allocating memory pages when they are accessed for the first time.

To better control data distribution on NUMA systems, we propose two different approaches :

- either the application programmer explicitly allocates data on specific NUMA nodes of the machine through a dedicated API we provide [7] (`omp_locality_domain_allocate_XXX`) where `XXX` may be a bloc cyclic data distribution for one or two-dimensional arrays over MAMI [2];
- or the application programmer only marks some regions of code that initialize data to give the runtime system the opportunity to map the corresponding tasks to make the first-touch allocation policy indirectly apply the data distribution we target. Indeed, Olivier et al. [11] have shown that specifying affinity for initialization tasks can lead to huge improvement over locality oblivious techniques. To avoid remote memory accesses, the threads must access the data during the computation phase the exact same way it was accessed during the initialization phase, which is very difficult to guarantee with dynamic task-based parallelism. We extend the OpenMP runtime in two ways. First, by adding functions to provide a dedicated API: `omp_set_affinity` to make the runtime map the next task to a specific NUMA node. Secondly, by extending scheduling heuristics to take into account task's dependencies to better map ready tasks.

During the application's execution, the runtime relies on system's `get_mempolicy` to determine on which physical node data are allocated. This information is then used to guide the way we perform task creation and load balancing.

3.3 Distribution of initial ready tasks : WSpush_init strategies

We refer to *initial tasks* when considering the sources of a task dependency graph, usually declared at the beginning of an OpenMP parallel region. These tasks are basically the first ones to be marked as ready and to be distributed over the platforms' places. We have implemented two initial tasks distribution strategies : `cyclicnuma` which distributes the tasks in a round-robin fashion over the NUMA nodes, and `randnuma` which randomly distributes the tasks over the NUMA nodes. Note that unlike `numactl`, the strategies we implemented consider the whole data appearing in the OpenMP task `depend` clause instead of working at the page level. In other words, while the two memory pages holding an 8K-wide array would be distributed on different nodes by `numactl --interleave=all`, they are always assigned to the same NUMA node when using one of our data distribution strategies.

3.4 Distribution of ready tasks : WSpush strategies

This section describes four different ways of pushing ready tasks to a NUMA system places. Two of them are data-oblivious while the other two rely on the dependencies expressed using the `depend` keyword on OpenMP tasks.

The `pLoc` strategy makes a processor push ready tasks to its own place, while the `pLocNum` strategy makes a processor push ready tasks to the place of its NUMA node (*local NUMA node*). The `pNumaW` strategy pushes tasks on the node-level place corresponding to the NUMA node where most of their output data are allocated to (W stands for Write). The last WSpush strategy, called `pNumaWLoc`, behaves almost the same than `pNumaW` except that if the data are allocated to the NUMA node of the processor pushing the task, we directly push the task to this processor's place instead of pushing it to the node-level place (Loc stands for Local).

It's important to note that `pLoc` and `pLocNum` does not take initial data placement into account, while `pNumaW` and `pNumaWLoc` are both aware of where a task's data are physically allocated and which of them are written, thanks to the OpenMP `depend` keyword.

3.5 Dynamic load balancing using work-stealing : WSselect strategies

Another important step when implementing work-stealing is the selection of the victim processor we want to steal from. This section describes the selection strategies we implemented, that take the architecture memory hierarchy into account. The first two strategies, `sRand` and `sRandNuma` are similar to those studied in [10] and distinguish two levels of hierarchy : the processor level and the NUMA node level. `sRand` selects a random processor's place while `sRandNuma` selects a random NUMA node's place. We additionally implemented several strategies mixing both levels of hierarchy, described below.

- `sProcNuma` : First, we browse the processor's place. Upon failure, we browse the topology in the following order : we first browse one of the neighbor processors ; when all the neighbors have been visited, we browse the local NUMA place ; we

continue by browsing all the processors' places from a random remote node and we eventually consider the place of its NUMA node.
- `sNumaProc` : This strategy is similar, except we always look at the NUMA place before looking at the processors' place.
- `sProc` : In this strategy the stealer will visit only the processors' places and its own NUMA place.
- `sNuma` : In this strategy the stealer will visit only NUMA places and its neighbors.

Like proposed in [11], all these strategies come in two versions : a `strict` version in which we prevent processors from stealing from other NUMA nodes to improve data locality and a `loose` version where these restrictions do not apply.

4 Evaluation

We ran all our experiments on the Intel192 machine described in section 2.1. We evaluated our strategies using the KASTORS [15][1] benchmark suite. More specifically, we used the dependent tasks version of the blocked QR factorization (`dgeqrf_taskdep`), and of the blocked Cholesky factorization (`dpotrf_taskdep`). These applications rely on kernels from BLAS and LAPACK libraries provided by OpenBLAS 2.15. We used the OpenMP GCC compliant runtime libKOMP [3] based on XKAAPI runtime system. We tagged the version we used on XKAAPI's git repository[2] in the branch *public/europar2016*. For all of the above applications, we used the GCC 5.2.0 compiler. We also made our execution log files public[3], as well as all the scripts we used, so that anyone may reproduce our data analysis, and look at the other results we did not put in the figures.

4.1 Impact of the data distribution

We first evaluated the impact the initial data distribution has on the application performance. We did an evaluation for multiple matrix sizes and block sizes, as well as multiple combinations of WSpush and WSselect strategies. Figure 2 reports the results we obtained for the Cholesky application, on 32K-wide matrices divided into blocks of 512×512 elements. We observed similar behavior running Cholesky on different matrix sizes (16K to 64K) and block sizes (256 to 1024). The lower double dashed horizontal line is the GCC performance baseline using sequential initialization. The middle dashed line is the same experiment using `numactl`. The upper solid line is the GCC baseline using parallel initialization.

Using `numactl` provides an important performance gain compared to the sequential initialization. However using a parallel initialization, either controlled (`cyclicnuma`, `randnuma`) or not (GCC init-para), is necessary to significantly improve the performances, regardless of the strategies used.

The `cyclicnuma` distribution is the one that works best regardless of the strategies, and we will use it as the default strategy for the next experiments.

[1] git available at https://scm.gforge.inria.fr/anonscm/git/kastors/kastors.git, tag "tag-europar16"
[2] https://scm.gforge.inria.fr/anonscm/git/kaapi/xkaapi.git
[3] https://github.com/viroulep/europar-2016-public

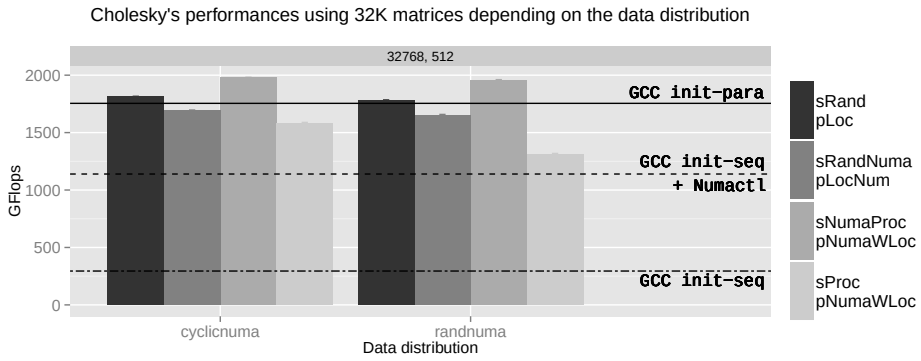

Fig. 2: Evaluating data distributions for multiple strategies (WSselect + WSpush)

4.2 Impact of the stealing restriction

Given a data distribution, previous works [11] have shown that restricting the task execution to the node where the data are written leads to better data locality which may improve the application performance. However, this is heavily dependent on the algorithm the application implements. For instance, in the case of a Cholesky factorization, many tasks write to the diagonal tiles of the matrix comparatively to other tiles of the matrix. Therefore applying a steal restriction on these tasks will potentially lead to an important number of inactive processors. We evaluated both strict and loose versions of our work-stealing strategies and found out that preventing processors from stealing from other NUMA nodes can lead to a loss of performance by around 25% to more than 75% with respect to the same setup without the *strict* restriction. For the sake of brevity we did not include a figure for this, but the results of these experiments are included in the logs publicly published.

4.3 Overview of the strategies performances

We took a given data distribution, `cyclicnuma`, and compared the different strategies, without any steal restriction. The performance obtained running the Cholesky application executed by the libGOMP [4] runtime system (without modification) is considered as a baseline for these experiments. Once again, even if the performances we obtained are obviously not the same, the behavior of the different strategies comparatively to each others are similar for the different applications we ran. Figure 3 shows the results of the experiments for the Cholesky application on 32K-wide matrices divided into blocks of 512×512 elements (best configuration for this matrix size). The dashed horizontal line is the GCC performance baseline using parallel initialization.

It is first interesting to note that even very basic `WSselect` and `WSpush` strategies, like `sRand`+`pLoc`, obtained decent performances thanks to the data distribution. Also, given a selection strategy (e.g. `sRandNuma`), placing the task on the NUMA

[4] GCC 5.2.0

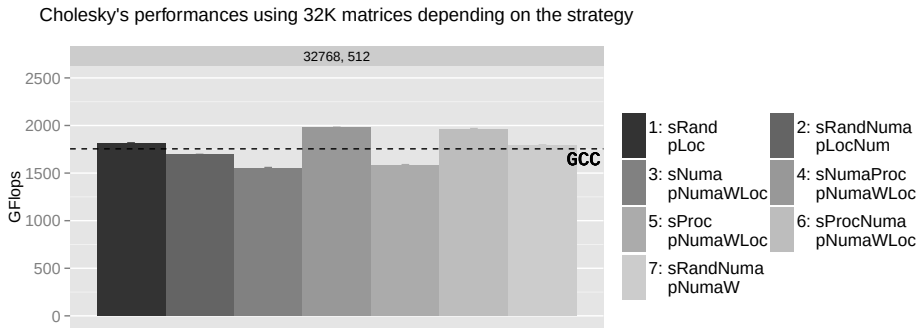

Fig. 3: Evaluating all strategies (WSselect + WSpush), using *cyclicnuma* WSpush_init

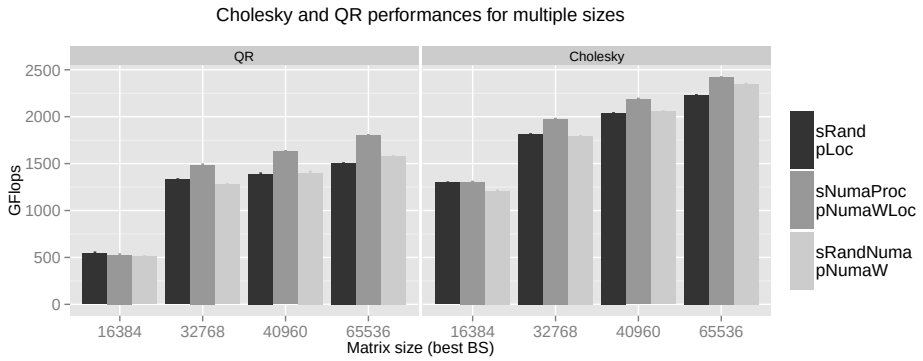

Fig. 4: Evaluating specific strategies on multiple sizes, using *cyclicnuma* WSpush_init

node where the written data are allocated (`pNumaW`) behaves better than simply pushing the data to its NUMA node (`pLocNum`). However, assuming the tasks are being pushed using the `pNumaWLoc` strategy, focusing the place selection on only one level of the hierarchy (`sProc` or `sNuma`) fails to reach the same level of performance we obtained with naive strategies. On the contrary, taking into account both levels of the hierarchy (`sProcNuma`, `sNumaProc`) achieve similar performance that outperforms other strategies.

4.4 Strategies performance scaling

We eventually selected the three strategy combinations that outperformed the GCC baseline to evaluate their scalability depending on the size of the input matrix. These strategy combinations are :

– `sRand` + `pLoc`, which is a basic strategy that does not take the architecture topology into account ;

– `sNumaProc` + `pNumaWLoc`, which was the best strategy in our previous evaluation and is also equivalent to using `sProcNuma` ;
– `sRandNuma` + `pNumaW` that performs random selection of node-level places.

Figure 4 reports their performances using a `cyclicnuma` distribution without steal restriction. The figure shows the performances using the best block size for each matrix size (which is, for our setup, 256 for a matrix size of 16384, and 512 for the others). As expected, combinations of strategies taking both the architecture hierarchy and data locality into account (`sNumaProc` + `pNumaWLoc`) achieve the best performances. The only exceptions are for small matrix sizes (16384), where there may just be not enough work to be able to take advantage of this strategy. We must note that simply distributing the data over the nodes enables the basic `sRand` + `pLoc` combination to achieve satisfying performances.

Finding the appropriate combination is highly application-dependent, therefore it is hard to give a solution for every type of application. However some general guidelines can be followed:

– The most critical part is the initial data distribution. Figure 2 showed it is absolutely necessary to use one.
– Hierarchical strategies have a cost, so the problem size has to generate enough work and data transfer to see a benefit.

5 Related work

Numerous works focus on data locality and/or topology-aware task scheduling strategies for NUMA architectures. Clet-Ortega et al. [5] studied different ways of decorating the architecture topology with task lists and how it impacts the performance of task-based applications on NUMA systems, promoting private per-threads lists of tasks browsed in a hierarchical way by work-stealing strategies. We somehow extended this work considering also node-level task lists. We showed considering these lists for pushing ready tasks and selecting work-stealing victims can help improving performance on NUMA systems. Olivier et al. [10] evaluated hierarchical task scheduling with respect to traditional centralized or distributed task schedulers. Creating a thread list, called *shepherd*, per NUMA node allowed their hierarchical scheduler to outperforms other approaches on several task-based applications. Tahan et al. [13] also studied the behavior of task-based OpenMP applications on NUMA systems, extending the NANOS runtime system with two NUMA-aware task schedulers called DFWSPT and DFWSRPT, taking into account the notion of task priority when pushing tasks to core-level queues. They also try to minimize the number of *memory hops* when performing load balancing. Drebes et al. [6] proposed similar ideas in another dataflow programming model named OpenStream. This model has a focus on data streaming and has a lot of flexibility on their placement, but does not provide flexibility to the user.

While the same kind of studies have been conducted in other contexts [14, 17, 16], none of them takes advantage of the OpenMP `depend` clause, which precisely indicates which data are read and written by a given task. As advertised by the results obtained by our `sNumaProc+pNumaWLoc` combined strategy, this information is worth taking into account when choosing a place to push ready tasks to.

6 Conclusion and future work

Task-based programming environments like OpenMP have become a standard way to program large-scale NUMA systems. Indeed, they give the programmer ways of expressing massive fine-grain parallelism that can be dynamically mapped to the architecture topology at runtime. OpenMP recently evolved to deal with tasks dependencies describing the data a task reads as input and writes as output.

This paper presented several runtime-level strategies to efficiently assign tasks to processors on any NUMA architecture. We presented strategies assigning ready tasks to lists of tasks, called *places*, attached to processors and NUMA nodes. These strategies define the way a task-based runtime system pushes ready tasks to their initial place and the way idle processors browse the architecture topology to select a place to steal from. We considered several initial data distributions and evaluated different combinations of "push" and "select" strategies on a 192 core NUMA system, on linear algebra applications. We achieved the best performance with strategies taking into account both the architecture topology and the initial data placement obtained through OpenMP tasks dependencies.

A short-term future work will be to extend an OpenMP compiler to be able to identify the initialization tasks in a more OpenMP-friendly manner, like extending the `task` construct with a `init` clause. We also intend to experiment with more OpenMP 4 applications. In a longer term, we intend to move our focus to compile-time techniques able to infer and to attach valuable information on tasks, like an estimation of a task operational intensity, that could guide some of the runtime system's decisions regarding task scheduling and load balancing. 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 software environment for very high performance computing.

References

1. R. Bleuse, T. Gautier, J. V. F. Lima, G. Mounié, and D. Trystram. *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, chapter Scheduling Data Flow Program in XKaapi: A New Affinity Based Algorithm for Heterogeneous Architectures, pages 560–571. Springer International Publishing, Cham, 2014.
2. F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst. ForestGOMP: an efficient OpenMP environment for NUMA architectures. *International Journal on Parallel Programming, Special Issue on OpenMP; Guest Editors: Matthias S. Müller and Eduard Ayguade*, 38(5):418–439, 2010.

3. F. Broquedis, T. Gautier, and V. Danjean. Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 102–115, Berlin, Heidelberg, 2012. Springer-Verlag.
4. F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In M. Danelutto, J. Bourgeois, and T. Gross, editors, *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2010, Pisa, Italy, February 17-19, 2010*, pages 180–186. IEEE Computer Society, 2010.
5. J. Clet-Ortega, P. Carribault, and M. Pérache. Evaluation of openmp task scheduling algorithms for large NUMA architectures. In F. M. A. Silva, I. de Castro Dutra, and V. S. Costa, editors, *Euro-Par 2014 Parallel Processing - 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, volume 8632 of *Lecture Notes in Computer Science*, pages 596–607. Springer, 2014.
6. A. Drebes, K. Heydemann, N. Drach, A. Pop, and A. Cohen. Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. *ACM Trans. Archit. Code Optim.*, 11(3):30:1–30:25, August 2014.
7. M. Durand, F. Broquedis, T. Gautier, and B. Raffin. In *Proceedings of the 9th International Conference on OpenMP in the Era of Low Power Devices and Accelerators*, pages 141–155, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
8. M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.
9. T. Gautier, X. Besseron, and L. Pigeon. Kaapi: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In *PASCO'07*, 2007.
10. S. Olivier, A. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins. Openmp task scheduling strategies for multicore NUMA systems. *IJHPCA*, 26(2):110–124, 2012.
11. S. L. Olivier, B. R. de Supinski, M. Schulz, and J. F. Prins. 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 '12, pages 65:1–65:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
12. OpenMP Architecture Review Board. OpenMP application program interface version 4.0, July 2013.
13. O. Tahan. Towards efficient openmp strategies for non-uniform architectures. *CoRR*, abs/1411.7131, 2014.
14. C. Terboven, D. Schmidl, T. Cramer, and D. an Mey. Task-parallel programming on NUMA architectures. In C. Kaklamanis, T. S. Papatheodorou, and P. G. Spirakis, editors, *Euro-Par 2012 Parallel Processing - 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings*, volume 7484 of *Lecture Notes in Computer Science*, pages 638–649. Springer, 2012.
15. P. Virouleau, P. Brunet, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, and T. Gautier. Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In *10th International Workshop on OpenMP, IWOMP2014*, pages 16 – 29. Springer, 2014.
16. T.-h. Weng and B. M. Chapman. Implementing openmp using dataflow execution model for data locality and efficient parallel execution. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 180–. IEEE Computer Society, 2002.
17. M. Wittmann and G. Hager. Optimizing ccnuma locality for task-parallel execution under openmp and TBB on multicore-based systems. *CoRR*, abs/1101.0093, 2011.