On the Impact of OpenMP Task Granularity

Thierry Gautier(✉), Christian Perez, and Jérôme Richard

Univ. Lyon, Inria, CNRS, ENS de Lyon, Univ. Claude Bernard Lyon 1, LIP, 69007 Lyon, France
thierry.gautier@inrialpes.fr, {christian.perez,jerome.richard}@inria.fr

Abstract. Tasks are a good support for composition. During the development of a high-level component model for HPC, we have experimented to manage parallelism from components using OpenMP tasks. Since version 4-0, the standard proposes a model with dependent tasks that seems very attractive because it enables the description of dependencies between tasks generated by different components without breaking maintainability constraints such as separation of concerns. The paper presents our feedback on using OpenMP in our context. We discover that our main issues are a too coarse task granularity for our expected performance on classical OpenMP runtimes, and a harmful task throttling heuristic counter-productive for our applications. We present a completion time breakdown of task management in the Intel OpenMP runtime and propose extensions evaluated on a testbed application coming from the Gysela application in plasma physics.

Keywords: Task granularity · Reordering · Cache reuse
Component model

1 Introduction

Tasks have been incorporated in OpenMP-3.0 in November 2008. This initial model only considers *independent* tasks, such as provided by the famous Cilk [13] parallel programming environment. In July 2013, OpenMP-4.0 integrates a *dependent*-task model. This model enable computing complex schedules that favor, for instance, data reuse among tasks.

One of our main testbed application extracted from the Gysela application [17] has been parallelized using dependent tasks. Preliminary experiments have shown that a hand-coded version of the code can greatly improve performances due to a better use of caches, but at the expense of code maintainability and, also with a loss of performance portability caused by hard-coded scheduling decisions. This paper reports our mitigated experience on delegating all task scheduling concerns to the OpenMP runtime. Our issues mainly come from the required fine-grain task granularity since reusing in-cache data is expected in our application.

B. R. de Supinski et al. (Eds.): IWOMP 2018, LNCS 11128, pp. 205–221, 2018.
https://doi.org/10.1007/978-3-319-98521-3_14

The algorithmic structure of the testbed application is the following: the working set is decomposed by planes, each plane is sub-divided in regions (such as line groups) where a chain of k tasks operate on it. The $k-1$ first tasks of each chain are independent, while the last task of each performs a per-plane stencil computation. Thus, tasks working on different planes are independent. Figure 4 illustrates it. At the end of each iteration, a final task operates on all regions of the same plane. Because the graph structure is quite simple, at the beginning of this work we were very confident to delegate the all task scheduling concerns to an OpenMP runtime.

Depending on the size of the working set and the hardware, only some regions or few planes could be fit into the shared cache. Two problems occur. First, the task creation iterates over all the first tasks of all chains, then over all the second tasks and so forth, which sequentially iterates several times over all the working set with causing $O(k)$ evictions. Second, the scheduling heuristics of the tested OpenMP runtimes (an Intel-based, LLVM and a GNU runtime) are not designed for constructive cache sharing. For instance, the Intel runtime relies on a work-stealing scheduler where working threads tend to have disjoint working sets. Constructive cache-sharing schedules have been studied since long time [7,10].

These two problems are strongly connected. The order of the task creation could not be easily chosen due to software engineering constraints. In our application, a high-level assembly of components [5] enforces the order and we do not want to violate the separation of concern by analyzing[1] memory access patterns arising from tasks submitted by different components. Moreover, even if we reschedule tasks in order to provide an efficient sequential execution, there is no guarantee that the OpenMP task scheduler will exploit it for constructive cache sharing.

Considering the scheduling performance guarantee as the most prominent issue, preliminary experiments of our application using OpenMP tasks enable us to locate four performance critical issues:

overhead: The task implementation has a significant overhead that limits scalability. In our case, it cannot be easily amortized by computation because of the fine granularity.

concurrency: The task creation is slowed down as the number of threads increases.

harmful heuristic: The task throttling [12] may improve performance. But a naive static heuristic is implemented on several OpenMP runtimes and it has been proved highly counter-productive. When present, the scheduler could not be clairvoyant on the future of the computation because almost all tasks are serialized.

task scheduling: Even if the task throttling is disabled, the default scheduling strategy between thread sharing cache favor, as discussed above, breadth-first execution where cores tend to have disjoint working sets.

In the case of coarse grain applications, the task creation overhead and concurrency issues are amortized by the computation [6]. The first three issues may

[1] Such analysis may be complex if made statically.

be overcome at the expense of a dedicated and optimized implementation: our experimental results with libKOMP, an extended version of the LLVM OpenMP runtime, illustrates the gains in term of performances. Nevertheless, the last issue is about scheduling where the best solution often relies on the application pattern. In this paper, we propose a two steps solution where submitted tasks are reordered cooperatively with the scheduler. This also points out one of the missing feature in OpenMP standard: the capability to specify specialized scheduling strategies for a set of tasks.

2 Background: OpenMP Tasks Management

This section deals with the LLVM OpenMP runtime [2] tag release 5.0 as currently developed by LLVM team[2]. It also compares some of the key design choices with those implemented in the GNU OpenMP libGOMP [1] coming with GCC 6 series.

2.1 Implementation of the OpenMP Task Model

The OpenMP task model enables the creation of tasks with dependencies in a simple way as sketched in the next listing.

```
#pragma omp task depend(inout: a, b) depend(out: c) depend(in: d)
    < code >
```

The encountering thread of the OpenMP task directive creates a task that could be performed asynchronously to the caller. A task execution corresponds to an execution of <code>. Data sharing attributes describe how the task data environment is built from the environment of the encountering thread.

The compiler and the runtime are responsible for the management of task internal data structures. For instance, the Intel and Clang compilers generate a pair of runtime calls [2] to `__kmpc_omp_task_alloc` and `__kmpc_omp_task`, for independent tasks, or `__kmpc_omp_task_with_deps` if the task directive includes depend clauses. The previous listing is translated to the following pattern (missing parameters are not important here) where two main function calls are marked in bold:

```
kmp_int32 outlined_function(kmp_int32 gtid, void* taskdata)
{  ... < code > ... }

kmp_tasking_flags_t flag = ...;
kmp_task_t* newtask = __kmpc_omp_task_alloc( ..., ..., &flags,
    size_of_task, size_of_shared, outlined_function, ...);
kmp_depend_info_t dep_list[4] = {..., ..., ..., ...};
__kmpc_omp_task_with_deps( ..., ...,  newtask, 4, dep_list, ..., ...);
```

The GNU compiler and libGOMP runtime merge these two calls [1] at the expense of recopying parts of the task data generated by the compiler on the C stack:

[2] https://openmp.llvm.org, http://llvm.org/git/openmp.git. The LLVM runtime has been forked from Intel public source and it is fully compatible with GCC, ICC and Clang compilers.

```
1 void outlined_function(void* taskdata)
2 { ... }
3
4 <opaque type> taskdata = { .... };
5 void* dep_list[5] = { 4, 3, a, b, c, d }
6 GOMP_task( outlined_function, taskdata, fcopy_taskdata, datasize,
7     ..., ..., flags, dep_list, ...);
```

Many other OpenMP runtimes follow the same approach: the compiler generates the code of the outlined function with correct copies or data sharing (according to the specified data sharing rules). Then, the runtime allocates an internal task descriptor, copies the fields, computes the dependencies and then pushes the task to various scheduling queue(s). The next section focuses on this internal data structure and algorithms used to build correct dependencies. Their choices explain the observed overhead or limitations.

Table 1. Main characteristics of libGOMP and libOMP. The sizes are in bytes and the task descriptors take into account structures for managing dependencies.

	Size of task descriptors	Dependencies	Task throttling threshold	Queues	Scheduler
libGOMP		Hash table + lock per team	64× number of threads	Per task (children), task group and teams	Multiple lists scheduling
libOMP	424	Hash table + lock per dependencies	256 tasks per queue	One per thread	Work stealing

2.2 Internal Data Structures and Algorithms to Manage Dependencies

GNU libGOMP and LLVM/Intel libOMP runtimes have made very different implementation choices as summed up in Table 1. The main difference between libGOMP and libOMP comes from the locking strategy to ensure coherent computation of dependencies: in libGOMP, exclusive accesses are guaranteed by a lock associated to the team data structure, while in libOMP, there is one lock per task. This explains scalability issues of libGOMP when the task granularity is too small [21,27].

The OpenMP dependent-task model is based on defining *dependence-type* of a list of memory references in the clause `depend`. The runtime should keep track of the previous accesses made on memory regions described by the array sections of the depend clause. Up to now, the standard restricts the usage to avoid the overlap of two array sections. This make the computation of dependencies much simpler in a sequence of tasks by identifying an array section to its base array. Indeed, runtimes can only store the last dependency into an associative table to retrieve it from a pointer. The task creation consists in the following steps:

Allocation of the internal task descriptor. libGOMP relies on the `malloc` function of the C library. The LLVM and Intel runtimes implement a thread-local heap allocator.

Initialization of the task descriptor fields. Once allocated, the runtime initializes a data structure, copy the ICVs, and update a counter to detect termination.

Checking dependencies. This step consists in adding the newly created task into the list of successors from all its predecessor. In the two runtimes, the scheme is almost the same: for each pointer identifying the array section, the runtime looks into a hash table to retrieve the last dependencies of the array section.

Enqueue. If a task is detected as ready for execution, then it is enqueued into runtime queues. The LLVM and Intel runtimes push a task into a queue owned by the running thread. GNU libGOMP enqueues the task into several queues: the child queue of its parent, the queue of the task group if it exists, and the queue of the team that stores all the ready tasks.

This high-level view masks the way the Intel, LLVM and GNU runtimes manage concurrency. The steps 'Allocation' and 'Initialization' are mostly involving local updates of data structures. They do not require locking mechanisms for exclusive accesses. Checking dependencies is the most complex operation of the task creation since predecessors of a task may finish while the task is being checked. The design of GNU libGOMP is such that all modifications related to dependencies are mutually exclusive by using a global lock associated with the team. This is the main scalability problem of libGOMP. The LLVM and Intel runtimes enable more concurrency between insertions and suppressions of dependencies. To manage the modification of data structures, they use a lock per *dependency node* attached to each task. Because concurrent accesses are more frequent, the thread generating tasks is slowed down: new tasks are created and enqueued at a low throughput compared to a sequential task creation.

2.3 Task Throttling

The term 'task throttling' refers to all kind of heuristics [4,12]. It enables the runtime to serialize tasks in order to reduce the inherent overhead of task creation. Sophisticated strategies have been designed and experimented [12] which dynamically profiles the application tasks to produce good decisions. In the LLVM and Intel libOMP or GNU libGOMP threads throttle task creations are based on static thresholds: when there is more than `256` tasks per queue in LLVM libOMP; and when there is more than `64 * omp_get_num_threads()` pending tasks in libGOMP.

These heuristics can efficiently reduce the overhead of task creation (see next section). However, these heuristics are not well suited, and even harmful, for some classes of applications [18], such as our. There is a huge gap between these research results and heuristics found in those OpenMP runtimes. Moreover, the scheduling decision could not be adapted during runtime.

3 Performance Evaluation and Extension of the LLVM Runtime

Experiments have been made on a quad-socket server with 4 NUMA nodes. Each NUMA node holds a 24-core Intel Xeon E7-8890v4 CPU for a total of 96 cores. The goal of the experimentations is to evaluate the capacity of fine-grained OpenMP tasks to be a building block to improve reuse of data in shared caches. We restrict all our experimentation on one NUMA node with up to 24 cores.

We make use of the LLVM libOMP version from http://llvm.org/git/openmp.git, branch `release_50`. The source code of the LLVM runtime has been instrumented to precisely measure the clock cycles for basic operations for the OpenMP task management in libOMP. We use the *time stamp counter* (`rdtsc`) that is incremented at constant rate on the platform.

3.1 Completion Time Breakdown of OpenMP Tasks Management

The LLVM OpenMP runtime libOMP has been instrumented to measure the delay for each the different steps in the task creation as presented in Sect. 2.2. In order to limit the overhead, we insert calls to get the real time stamp counter a the begin and the end of each of these steps. Delays are cumulated per thread and a final summation is computed at the end of the program to avoid overhead due to concurrent update. It impacts six functions, including the initialization of finalization of the library to dump the values.

Figure 1 reports results for the BOTS [11] benchmarks with only independent tasks. Figure 2 reports results on the Jacobi and SparseLU benchmarks of the KASTORS suite [27]. They compare two versions of the same code: one with independent tasks and the second with dependent tasks.

Each measure is the average cycles per operation over 30 runs. In all figures, we present the number of cycles for the following internal operations: `alloc` is the allocation and initialization of the data fields for the internal task descriptor; `atomic` is extracted from `alloc` and refer to a piece of code that update concurrent object by atomic instruction; finally, `enqueue` is the operation of inserting the descriptor into a scheduler queue. On the benchmark with dependent tasks, `check deps` is the operation of checking and adding the dependencies between tasks and `release deps` is the operation of releasing successors of the ended tasks. The sum of all these operations captures the code between a task submission and its insertion in scheduler queues.

For the independent task benchmarks, the serialization of all submitted tasks on the case of 1-core execution shows that a task throttling heuristic can reduce the overhead of task management. The task initialization cost increases slowly as the number of cores grows: parts of the initialization make use of atomic operations for which the cost depends on the number of concurrent data accesses. The enqueue operation is stable mostly when the number of cores is greater than 1, except for Uts [18] which is a search algorithm working on very large unbalanced trees: the concurrency on each queue of libOMP is exacerbated.

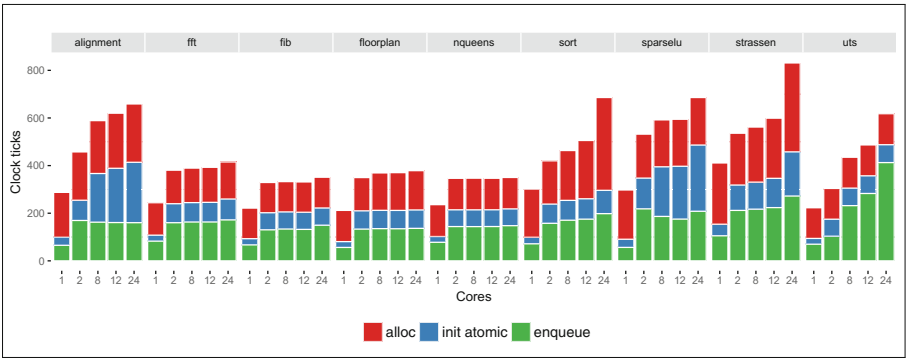

Fig. 1. Completion time breakdown of OpenMP independent tasks in libOMP on BOTS.

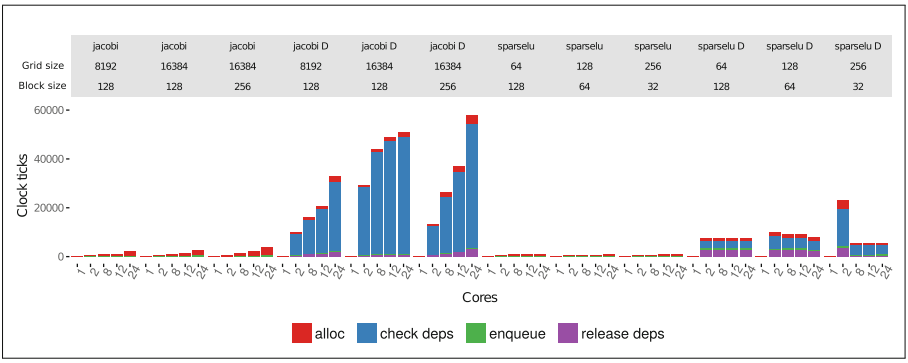

Fig. 2. Comparison of the completion time breakdown between OpenMP-3.0 tasks and OpenMP-4.0 dependent tasks on the Jacobi and SparseLU benchmark from the KASTORS. The suffix 'D' denotes the dependent task version of the code.

For the KASTORS benchmark, except for Jacobi, the global behavior is similar to SparseLU in Fig. 2. On average, the cost of task creation is about 10 times bigger than for independent task. Most of the cost comes from checking dependencies. Next comes the release of dependencies to activate successors when tasks are finished.

Jacobi is a 2D stencil. The grid size is either 8192 or 16384 and the block size is 128 or 256. The application is memory bound and the tasks are very fine-grained (about 5×10^5 clock ticks). Concurrent data structures are under pressure because workers end their tasks quickly. It explains the big increase in task creation cost (Fig. 2 jacobi_taskdep for all the inputs) wherein the generating thread run in quasi-concurrence with one of the $P - 1$ other threads.

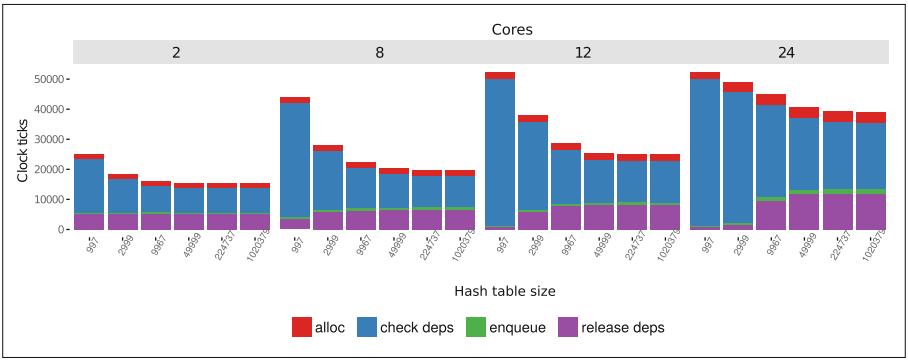

Fig. 3. Completion time breakdown of the jacobi_taskdep benchmark for different hash-table sizes (x axis). The standard hash-table size in libOMP is 997. The groups refer to the number of cores used.

3.2 Impact of the Task Serialization

In the LLVM libOMP, the task queues are bounded to 256. When the queue is full, the task throttling forces the serialization of the newly created tasks. Such a situation arises when the generating thread creates tasks faster than the worker threads can consume them. Increasing the queue size may impact the scheduling order of the tasks. For instance, in jacobi_taskdep [27], the generating thread creates first a set of independent tasks to copy an old data version in the new data version, then it creates tasks making stencil computation from an old data version to produce a new version. Tasks of the second set depend on tasks of the first set. In this case, the task throttling may block generation of tasks of the second set: the worker thread may not activate the successor tasks because they are not yet submitted!

On jacobi_taskdep and on the smallest grid (8192, blocksize = 128), we observe between 15% to 25% of gains for a range of a number of cores without task serialization (a queue of size 2^{16} is large enough). For a grid of size of 16384 and with the same block size (generating 4 times more tasks), the gain ranges from 2% on 24 cores to 19% on 2 cores (15% on 8 cores) with a small standard deviation.

3.3 Impact of the Hash Table Capacity

The hash table converts memory addresses to meta data in the libOMP procedure to compute dependencies (`__kmp_process_deps`). libOMP implements a hash table with separate chaining when keys are hashed to the same slot. When the load factor of the hash table increases, the cost of insertions becomes linear in the number of chained keys. If the number n of dependencies is high, the cost of finding a key is on average $O(n/s)$ where s is the number of slots.

By default, the number of slots in libOMP is 997 for each implicit task (which generally creates more dependent tasks). With this condition, the load

factor of the hash table is near to 1: almost all insertions cause hash collisions. We experiment `jacobi_taskdep` on a grid size of 16384 with a small block size of 128 and with different sizes of the hash table. The number of dependencies to resolve is 2883584. For sizes bigger than 49999, the gain is small. The completion time on 24 cores is 3.16 s with the default value and 2.21 s with a hash-table size of 49999: the gain on the completion time reach 30%. Figure 3 reports the completion time breakdown of the internal task management. As expected, the cost of checking dependencies is reduced as the hash table is getting bigger.

4 Evaluation of the Gysela Testbed Application

The evaluated testbed application is a prototype of semi-Lagrangian 2D advection extracted from Gysela, an iterative gyro-kinetic simulation of magnetic fusion plasmas [17]. The extracted part is the most computationally intensive of the whole application and improving its performance is a major concern. The prototype makes the uses of task-based scheduling since it offers a promising approach to improve the performance of the existing code (based on OpenMP fork-join directives) through a better data locality and a finer-grained parallelism.

4.1 Overview

Being able to maintain the application is crucial since several algorithmic variants are provided and new algorithms are regularly devised. While studying this aspect is beyond the scope of this paper, it deeply impacts the evaluated code. Indeed, the prototype is split into independent computational parts called software components [23] in such a way parts can be easily replaced. Components are then assembled during a compilation process [5] that produces an OpenMP code.

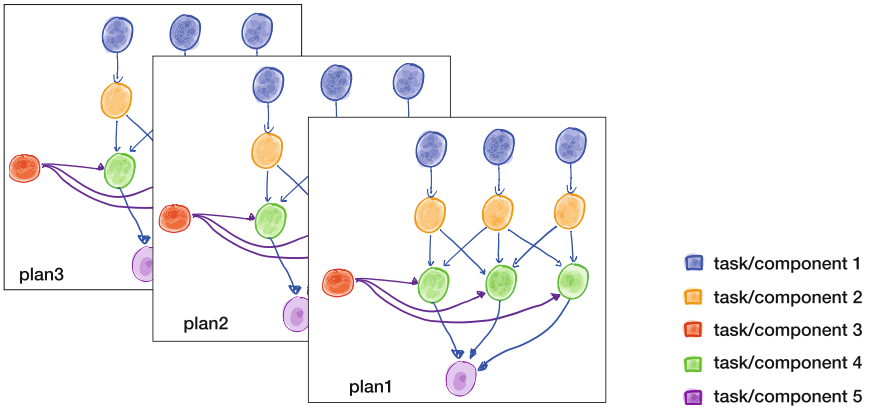

Fig. 4. Sketches of dependencies between OpenMP tasks in our testbed application. It represents tasks working on three planes. Each application level component spawns tasks for all the planes. (Color figure online)

The prototype iterates several times over 2D slices (plane) of 3D and 4D arrays. An iteration is defined by a sequence of 5 components. Each component generates a bag of independent tasks (following an SPMD approach) working on sub-parts of the planes (usually few lines). Assembling components results in adding dependencies between the generated bags of tasks. Figure 4 displays the structure of the task graph submitted to the runtime per plane of the working set. Tasks that work on different planes are totally independent. Because the graph structure is quite simple, at the beginning of this work we were very confident to delegate the all task scheduling concerns to an OpenMP runtime. However, performance issues have been identified on current OpenMP runtimes.

4.2 Task Submission

A carefully hand-written OpenMP-native implementation has been designed to study how fast it can be to use OpenMP tasks when all maintainability constraints are skipped. This implementation submits tasks by following a depth-first strategy, making use of recursive tasks (enabling parallel submission of independent tasks) and synchronization steps (enforcing runtimes to work on a sliding window of tasks). This implementation is 38% faster thanks to a better tasks scheduling and data-reuse in caches.

Although the hand-written implementation has demonstrated the feasibility in term of performance, important concerns such as code readability and separation of concerns are totally ignored. By using a HPC component model [5], the whole task graph is submitted sequentially all at once using a breadth-first strategy, as for the jacobi_taskdep benchmark. In practice, Component 1 submits a bag of many tasks, then Component 2 do the same and so forth.

It is worth noting that such a design comes from maintainability constraints. Indeed, the separation of concerns that helps to maintain components also hinders the use of a depth-first submission strategy. Moreover, it also prevents components to make assumptions on the implementation of other components, such as the dependency of submitted tasks. Since OpenMP 4.5 provides no way to submit dependent tasks in parallel, submission is doomed to stay sequential. Nevertheless, this design suffers from several sources of slowdown with both GNU libGOMP and LLVM libOMP and shared common conclusions with previous sections.

4.3 Characteristics of the Performances Drop

As for `jacobi_taskdep`, task submission becomes slower than the actual execution of tasks before they can be fine enough for the computation to fit better in caches resulting in starvation of worker threads and higher completion times. This high overhead comes from a combination of many technical factors: a small fixed-size hash table not well-suited for so many tasks, a contention of shared data structures in runtimes as tasks are being submitted while others are running.

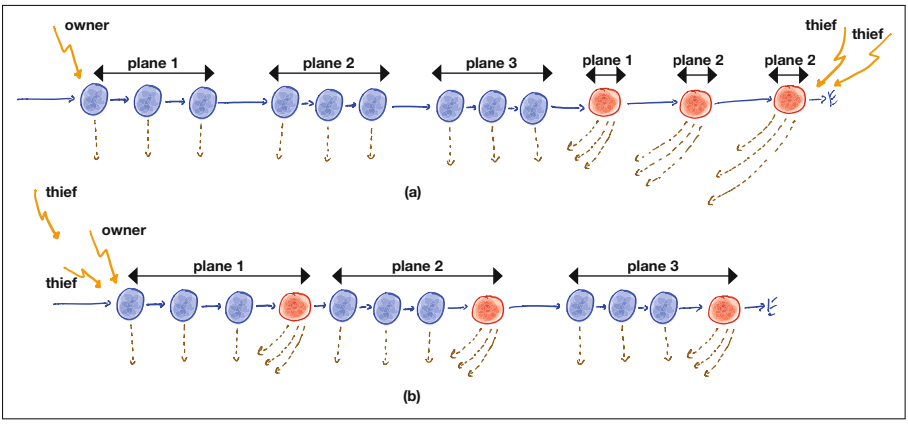

Fig. 5. Reordering strategy principle. (a) State of the ready list with the original work stealing after the task submission from components 1 (blue), 2 (hidden because tasks are dependent of the component 1) and 3 (red). See Fig. 4. (b) State of the ready list using the reordering strategy. (Color figure online)

The task throttling prevents the execution of tasks using a depth-first strategy as shown in Sect. 2.3: the submission is halted and current tasks executed before dependent tasks can be submitted. Without clairvoyance on all the computation, the execution order is close to the sequential order of task creation: this breadth-first strategy causes tasks to work simultaneously on a bigger amount of data (all the planes) resulting in poor utilization of caches.

4.4 Improving Locality Through Tasks Rescheduling

Even when the task throttling threshold is increased, the available scheduling algorithms are not able to group the execution of tasks working on the same plane although they are dependent and share data. The execution order mainly follows the submission order which turns out to be inefficient in our case. Figure 5(a) represents the submission order in the scheduler's ready list of the generating thread and the way the owner thread and thieves operate on the list during a steal operation. In work stealing, the owner (victim) and the thieves operate at the two extremities of the list to avoid any contention.

However, here, we want the cores to share data in caches. It is preferable that all threads operate on the same side of the list to favor data sharing. Thus, the LLVM libOMP function `__kmp_steal_task` has been modified to work in cooperation with functions that enqueue and dequeue tasks for the owner thread of the queue. Now, a thread enqueues new ready tasks at the same side of the list, where all other threads are working.

Keeping lists ordered as in case (a) is not enough, the ready tasks (red tasks of Fig. 4) have to be enqueued close to those working on the same plane. Thus, we have developed a fast reordering strategy of the ready list which computes

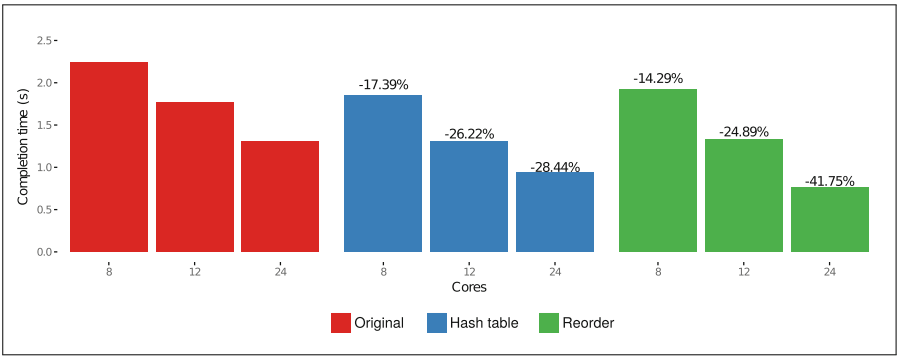

Fig. 6. Comparison of completion times for the Gysela application with different configurations.

on-line position where to insert ready tasks. This helps to favor the case (b) of Fig. 5. The heuristic is simple and well-suited for such a dependency task graph. It adds $O(1)$ instructions per dependencies.

Each task keeps the range of tasks in the ready list on which it depends. A task having no predecessor task is enqueued in the ready list and it initializes the range on itself. The algorithm which computes dependencies visits, for each newly created task, all its predecessors. During this step, the union of the range of all predecessors is incrementally computed, and the last inserted tasks in the ready list to the oldest in the union range is reordered. Due to dependencies, the ranges tend to include all the tasks. The reordering is currently stopped when the ranges become too wide.

Figure 6 reports the completion time on 8, 12 and 24 cores of the Gysela testbed application. A bigger hash-table size (132069 in place of 997) improves the performance by at least 17%. On 24 cores, the reordering achieves a performance gain of 41% over the original LLVM libOMP library.

5 Discussion

The OpenMP standard becomes predominant in the HPC runtime community. The recent integration of tasks into the standard has completely changed the way applications can describe parallel algorithms, enabling the description of more complex and finer-grained parallel computations. However, we are facing issues where OpenMP specification does not help us to guarantee performance portability. Indeed, in our case based on the decision to delegate the task management to OpenMP, the task granularity is enforced to reach high performance, while the task submission order is a consequence of the need for separation of concerns in the code. These two factors are the main sources of the issues explained in this paper: the task implementation of experimented runtime exhibits a high overhead and the submission order is not well-suited for reusing cached data. It

seems a better long-term solution to improve OpenMP rather than handcrafting the generated OpenMP code.

What solutions are offered by OpenMP? Let us consider several opportunities to solve our issues.

5.1 Optimizing OpenMP Runtime Implementation

There are technical solutions for some issues presented above. The first one concerns the task throttling heuristic which is too basic in LLVM and GCC runtimes. One way is to integrate a more robust heuristic, for instance, such as in [12]. Another possible direction would be to claim that such heuristic will potentially always takes wrong decisions, as in our case, with a strong performance loss. Note that in our past work [9], thanks to a very low overhead in task creation, our implementation, without any throttling heuristic, was very competitive with the GCC or Intel implementations. We think that the task granularity is an algorithmic parameter and that OpenMP provides an explicit way to control it using the clause `if` of the task directive. Thus, in our point of view, it is better to disable any throttling heuristic in the runtime that may impact performance, even if it is in few cases such as ours.

Another important parameter which impacts performances is the cost of finding dependencies using the hash table. Preliminary results for GCC exhibits a similar behavior. The LLVM runtime has a too small hash-table that, indeed, generate a lot of hash collision. This problem should be studied and we currently integrate in the LLVM runtime a resizable hash table (the size depends on the load factor).

5.2 Parallelization of Task Submission

As described in Sect. 4.3, if the task submission is slow compared to the execution, the scheduler may never be able to activate the dependent tasks because they are not yet created: the scheduler is not clairvoyant. A straightforward idea is to make the task submission parallel. As for Gysela, a simple way would be to take into account the independence of tasks that belong to different planes. However, the component model used need to be extended to take into account the hierarchical structure of some applications such as Gysela and the high-level component assembly compiler back-end need to be changed too.

Moreover, according to the current OpenMP standard, the parallel submission is restricted to independent tasks only. The enforced constraint on the depend clause [8] is that it "*establishes dependences only between sibling tasks*", *i.e.* between tasks that are child tasks of the same task region.

Past projects have deals with a way to parallelize task submission in presence of dependencies. For instance, Athapascan-1 [14] was able to successfully parallelize the task graph submission of a stencil [22] on distributed architectures using a *postponed access mode* in order to delegate real access to data to sub tasks. More recently, a similar solution proposed for OpenMP with the use weak

dependencies [20] seems very interesting if implementation scale enough with the number of submitted tasks.

5.3 Specialization of Task Scheduler

It is generally accepted that task scheduling depends on the targeted application. How to specialize a task scheduler for an OpenMP program? In addition to its original implementation, Cilk [13] provides guarantees on the expected performances in term of *work* and *depth* or *critical path*. What could be such a performance model for OpenMP task schedulers, even in presence of restrictions?

We propose a two steps organization of the way applications may influence the task scheduler of an OpenMP runtime. First, hints should be pass to the runtime in order to schedule a group of tasks according to a specific heuristic. Similarly to the clause `schedule` available for work-sharing loops, we expect a clause `task_schedule` for task groups and parallel directives. Such a clause enable the application to pick a specific task scheduler (among those provided) that should be preferred by the runtime, and may be defined by some expert users.

Secondly, in the same way OMPT has been defined to capture (in a portable manner) the state and the events generated by OpenMP runtimes, we expect to have access to an API (for experts) to enforce actions made by the runtime in order to have a better control over the scheduler or to redefined it.

6 Related Work

Optimizing task submission has been the subject of numerous works. In lazy approaches, the task creation is delayed until an idle resource requires tasks [16, 24]. Compilation strategies can reduce the overhead by exploiting the structure of the scheduler: for instance, the Cilk compiler generates two variants of each task (fast and slow clones) [13] in a way that move overheads out of the work and onto the critical path. However, this method, defined in Cilk as the *work-first principle*, may come at the expense of an impaired scalability. In [4], the authors have similar considerations about the generation of fast/slow clones. Orthogonal optimizations concern the optimization of the data-structure representation. The size of internal descriptors of the dependent tasks in LLVM libOMP is at most of 424 bytes per dependency, while in libKOMP, a native task descriptor is less than *64 bytes* explaining most of the speedup [9].

Swann [25] compared different methods of dependency analysis. TurboBLYSK [21] has proposed a way to cache dependencies of task graphs in order to reuse them without any overhead during the resolution. Following the work-first principle, in [15], the computations of dependencies have been moved from the work to steal operations.

A fast task creation can reduce the inactivity of worker threads. The scheduling algorithm may have a strong impact on the overall performance, such as the reorder method proposed in Sect. 4.4. A lot of scheduling heuristics in runtime

systems has been proposed to improve the task locality [3,7,10] and to control the task affinity [19,26], but few of them are dealing with task reordering as presented above.

7 Conclusion

This paper has presented preliminary reports of using fine-grained tasks in OpenMP. Most of the measures and developments have been made with the LLVM OpenMP runtime supported by the LLVM group. Due to the fine granularity and preliminary experiments, we assumed that GNU libGOMP would behave the same way with at least similar overheads. The completion time breakdown analysis has focused on the task submission, especially costs related to checking dependencies and in the way to make the scheduler clairvoyant in order to reorder the on-line queue of ready tasks.

Further investigations on a wider range of applications are needed for the reordering method.

Several extensions of the Intel libOMP have been proposed and implemented. Results obtained on the Gysela prototype are satisfactory. Future works will focus on optimizing an OpenMP runtime for issues identified by such an application: support for fine-grained task. Finally, if the overhead cannot be avoided, then parallelizing the submission may be a solution.

References

1. GNU libgomp. https://gcc.gnu.org/onlinedocs/libgomp
2. Intel®OpenMP* Runtime Library (2016). https://www.openmprtl.org
3. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2000, pp. 1–12. ACM, New York (2000)
4. Agathos, S.N., Kallimanis, N.D., Dimakopoulos, V.V.: Speeding up OpenMP tasking. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 650–661. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32820-6_64
5. Aumage, O., Bigot, J., Coullon, H., Pérez, C., Richard, J.: Combining both a component model and a task-based model for HPC applications: a feasibility study on gysela. In: Proceedings of GCCGrid 2017. IEEE (2017)
6. Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., Teruel, X.: An experimental evaluation of the new OpenMP tasking model. In: Adve, V., Garzarán, M.J., Petersen, P. (eds.) LCPC 2007. LNCS, vol. 5234, pp. 63–77. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85261-2_5
7. Blelloch, G.E., Gibbons, P.B., Matias, Y.: Provably efficient scheduling for languages with fine-grained parallelism. J. ACM **46**(2), 281–321 (1999)
8. OpenMP Application Review Board: OpenMP application programming interface - version 4.5, November 2015. https://www.openmp.org
9. 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). https://doi.org/10.1007/978-3-642-30961-8_8

10. Chen, S., et al.: Scheduling threads for constructive cache sharing on CMPs. In: Proceedings of SPAA 2007, pp. 105–115. ACM, New York (2007)
11. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP tasks suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: Proceedings of ICPP 2009, pp. 124–131. IEEE (2009)
12. Duran, A., Corbalán, J., Ayguadé, E.: An adaptive cut-off for task parallelism. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008, pp. 36:1–36:11. IEEE Press, Piscataway (2008)
13. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. SIGPLAN Not. **33**(5), 212–223 (1998)
14. Galilée, F., Roch, J.L., Cavalheiro, G.G.H., Doreille, M.: Athapascan-1: on-line building data flow graph in a parallel language. In: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, PACT 1998, pp. 88–95. IEEE Computer Society, Washington, DC (1998)
15. Gautier, T., Besseron, X., Pigeon, L.: KAAPI: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PASCO 2007 (2007)
16. Goldstein, S.C., Schauser, K.E., Culler, D.E.: Lazy threads: implementing a fast parallel call. J. Parallel Distrib. Comput. **37**(1), 5–20 (1996)
17. Grandgirard, V., et al.: A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations. Comput. Phys. Commun. **207**, 35–68 (2016)
18. Olivier, S., et al.: UTS: an unbalanced tree search benchmark. In: Almási, G., Caşcaval, C., Wu, P. (eds.) LCPC 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72521-3_18
19. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Prins, J.F.: Scheduling task parallelism on multi-socket multicore systems. In: Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers, ROSS 2011, pp. 49–56. ACM, New York (2011)
20. Pérez, J.M., Beltran, V., Labarta, J., Ayguadé, E.: Improving the integration of task nesting and dependencies in OpenMP. In: IPDPS, pp. 809–818. IEEE Computer Society (2017)
21. Podobas, A., Brorsson, M., Vlassov, V.: TurboBŁYSK: scheduling for improved data-driven task performance with fast dependency resolution. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 45–57. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11454-5_4
22. Revire, R.: Scheduling dynamic task graph on large scale architecture. Ph.D. thesis, Institut National Polytechnique de Grenoble - INPG, France, September 2004. https://tel.archives-ouvertes.fr/tel-00010909
23. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
24. Traoré, D., Roch, J.-L., Maillard, N., Gautier, T., Bernard, J.: Deque-free work-optimal parallel STL algorithms. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 887–897. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85451-7_95
25. Vandierendonck, H., Tzenakis, G., Nikolopoulos, D.S.: Analysis of dependence tracking algorithms for task dataflow execution. ACM TACO **10**(4), 61:1–61:24 (2013)

26. Virouleau, P., Broquedis, F., Gautier, T., Rastello, F.: Using data dependencies to improve task-based scheduling strategies on NUMA architectures. In: Dutot, P.-F., Trystram, D. (eds.) Euro-Par 2016. LNCS, vol. 9833, pp. 531–544. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43659-3_39
27. Virouleau, P., et al.: Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 16–29. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11454-5_2