Enhancing OpenMP Tasking Model: Performance and Portability

Chenle Yu[1,2(✉)], Sara Royuela[1(✉)], and Eduardo Quiñones[1(✉)]

[1] Barcelona Supercomputing Center, Barcelona, Spain
{chenle.yu,sara.royuela,eduardo.quinones}@bsc.es
[2] Universitat Politècnica de Catalunya, Barcelona, Spain
chenle.yu@upc.edu

Abstract. OpenMP, as the *de-facto* standard programming model in symmetric multiprocessing for HPC, has seen its performance boosted continuously by the community, either through implementation enhancements or specification augmentations. Furthermore, the language has evolved from a prescriptive nature, as defined by the thread-centric model, to a descriptive behavior, as defined by the task-centric model. However, the overhead related to the orchestration of tasks is still relatively high. Applications exploiting very fine-grained parallelism and systems with a large number of cores available might fail on scaling.

In this work, we propose to include the concept of Task Dependency Graph (TDG) in the specification by introducing a new clause, named *taskgraph*, attached to *task* or *target* directives. By design, the TDG allows alleviating the overhead associated with the OpenMP tasking model, and it also facilitates linking OpenMP with other programming models that support task parallelism. According to our experiments, a GCC implementation of the `taskgraph` is able to significantly reduce the execution time of fine-grained task applications and increase their scalability with regard to the number of threads.

Keywords: OpenMP specification · Tasking model · Runtime overhead

1 Introduction

OpenMP is a parallel programming model widely used on shared memory systems by virtue of its programmability, portability, and competitive performance. OpenMP 3.0 introduced support for fine-grained irregular parallelism with the so-called task-centric model. Later, OpenMP 4.0 introduced fine-grained data-driven synchronization mechanisms in the form of task dependencies. Since this preliminary support for task parallelism, the OpenMP specification has evolved from a *prescriptive* to a *descriptive* paradigm, enabling users to define what has to be parallelized rather than how to parallelize it. Interestingly, the tasking model can be now used not only for *task parallelism*, by using the `task` construct, but also for *data parallelism*, by using the `taskloop` construct.

S. McIntosh-Smith et al. (Eds.): IWOMP 2021, LNCS 12870, pp. 35–49, 2021.
https://doi.org/10.1007/978-3-030-85262-7_3

Despite the clear benefits of the tasking model (including flexibility, dynamism, and independence from the underlying resources), the implementations of this model typically introduce a considerable overhead related to the management of the parallel execution of tasks. As a result, these overheads have been extensively studied [8,13,15], concluding that a sufficiently coarse granularity of tasks (i.e., workload assigned to a task) is the keystone to obtain the expected performance gains. However, the smaller the granularity is, the greater the overhead will represent the end-to-end execution time. Although the runtime overhead is substantially dependent on each particular implementation, the observations made on the studies are independent of the compiler and the runtime used in the experiments.

To overcome the limitations derived from classic OpenMP implementations, different works propose alternative solutions. Castelló et al. presented an implementation using lightweight threads (LWT) instead of POSIX threads [2] and G.Tagliavini et al. designed and implemented an OpenMP runtime environment specifically for the Kalray MPPA 256 [5], a many-core processor for embedded systems. Despite the effectiveness of these solutions, they are either difficult to apply on mainstream OpenMP runtime implementations, or they are not portable to diverse shared memory systems.

This paper takes into account the limitations of the previous solutions, and proposes a new feature in the OpenMP specification to allow users to define *taskgraph* regions, i.e., regions of an OpenMP task-based program that can be implemented more efficiently. This enhancement in the implementation is substantiated on the Task Dependency Graph (TDG) used to represent the execution of a task-based program (or region), and is only possible if either (a) the TDG of the selected region can be completely expanded at compile-time (i.e., all tasks instances and their dependencies can be decided statically), or (b) the region is going to be executed multiple times and the same TDG can be exploited several times.

The approach proposed can reduce, by design, the runtime overhead related to task management, and it has a higher abstraction level than previously presented methods. Hence, existing OpenMP implementations, including aforementioned work, can easily integrate taskgraph, thus benefiting from several layers of optimization. Our main contributions are the following:

- A new approach for accelerating the OpenMP tasking model by reducing task runtime overhead, together with the analysis of the use cases that can benefit from this new approach.
- A new clause, namely `taskgraph`, providing the syntax and the semantics thereof, and the characterization of the implications on the execution and memory models of OpenMP.
- Preliminary results on the benefits that can be extracted from this new feature considering (a) performance gain by virtue of a lighter implementation of the OpenMP runtime, and (b) interoperability provided by the TDG, which allows using OpenMP as a high-level API that can be lowered to different programming models.

2 Motivation

The main sources of overhead in the OpenMP runtime for handling tasks include: (a) the contention caused by different threads accessing simultaneously to shared resources, for instance, acquiring the lock that protects a shared data element; and (b) the cost of handling tasks, including task creations, dependency resolutions and task deletions. While the former is proportional to the number of threads running concurrently and the amount of parallelism exposed in the application, the latter scales with the number of tasks, which tends to be large in modern HPC applications. This can be explained by the increased workloads and the growing number of logical threads incorporated in high-end modern processors.

On the whole, to achieve the levels of performance provided by modern multi-core and many-core accelerated architectures, the number of tasks exposed in an application must be, at least, as large as the number of threads during most part of the execution. However, using more threads does not systematically mean higher performance, according to the sources of overhead stated above. Therefore, reducing task-related overhead is of paramount importance for the success of OpenMP task-based frameworks.

In a perfect world, where the compiler can statically determine the data associated to all task instances and all dependencies among tasks, the allocation of tasks and the resolution of the dependencies can be done at compile time. We use an in-house implementation in the GCC 7.3.0 framework that uses a pre-computed Task Dependency Graph to allocate tasks and decide dependencies beforehand in order to illustrate the benefits of this approach. Figure 1 shows the execution time[1] of an optimized heat transfer simulation where the problem size is fixed (2048×2048 matrix), and the block size is changed throughout the experiment, generating from 640 tasks (with 256×256 block size) to 16000 tasks (with 52×52 block size). The line annotated as $GCC + taskgraph$ corresponds to the optimal case where the TDG is fully pre-calculated at compile-time, whereas the line annotated as $GCC + original\ GOMP$ runs the vanilla implementation of GCC GOMP runtime. Essentially, the figure shows when the number of tasks increases (and so the granularity of the tasks decreases because of the fixed total workload) the modified version does not lose performance, while the original version does once the number of tasks exceeds 4000.

Applications in the HPC domain are however commonly dynamic, in the sense that their data is only known at runtime. As a consequence, compilers are not able to automatically apply the optimizations explained above. However, some HPC applications show other patterns that can also benefit from a similar approach to reduce overhead. This is the case of applications that expose multiple levels of parallelism, where the outer levels are dynamic and the inner levels are static, e.g., the sLASs linear algebra solver [19], the Specfem3D simulator [6]

[1] The execution has run in a node of the Marenostrum IV [1] supercomputer, equipped with an Intel Xeon Platinum 8160 CPU, having 2 sockets of 24 physical cores each, at 2.1 GHz and 33 MB L3 cache.

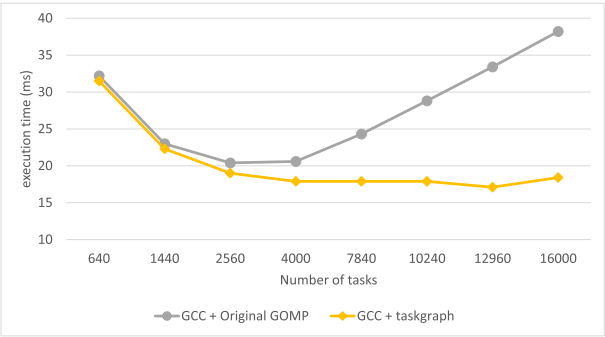


Fig. 1. Execution time of Heat transfer simulation using Gauss-Seidel method while changing the number of tasks and reducing the task granularity.

and the Quantum ESPRESSO material modeling tool [4]. In these cases, where inner TDG can become static after their first execution, benefits similar or even better than the ones shown in Fig. 1 can be expected.

Nonetheless, it is unattainable for a compiler to detect these cases and lower the code accordingly so the runtime does not create and destroy the inner (and stable) TDGs each time they have to run. As a consequence, this paper proposes to enable programmers to explicitly define the regions of their applications that are static (i.e., decidable at compile-time) or stable (i.e., these will run several times without changing the TDG and consumed data). In order to fit in the OpenMP specification without introducing unnecessary changes, we propose a new clause called `taskgraph` which, together with the `task` and `target` directives, acts as a hint to the compiler and the runtime system to recognize the task region to optimize.

Furthermore, the possibility of pre-building a TDG opens the door for programs to be lowered, not only to the common OpenMP runtime (e.g., GCC, LLVM), but also to other APIs in order to exploit the heterogeneity. This is of particular interest in modern supercomputers as, for instance, 6 of the 10 most powerful supoercomputers in the world now incorporate Nvidia GPUs to scale their computation power [18], and various applications are legacy code, or are highly tuned for a specific accelerator device. Thus, increasing the portability of OpenMP to these models, as well as enhancing the programmability of low-level APIs is crucial. Previous works [20] have already tackled this issue, and a detailed analysis on these aspects is further provided in Sect. 4.2.

3 The Taskgraph Model

Despite the fact that application developers often use TDGs to express and study their programs, the OpenMP specification does not include the concept of TDG per se. We propose to introduce this concept, named *taskgraph*, in the OpenMP specification to tackle the challenges mentioned in Sect. 2. To do so,

this section presents the *taskgraph* mechanism and discusses how to integrate this feature into the current OpenMP specification. Concretely, it defines first the syntax of a new `taskgraph` clause and its semantic, considering its impact on the execution and memory models of OpenMP; and then exhibits the conditions required by the OpenMP program to use the *taskgraph* feature correctly.

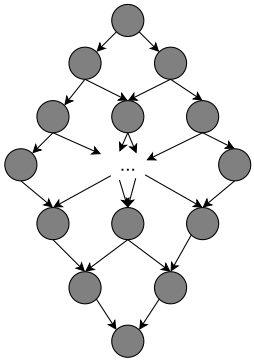
3.1 The `taskgraph` Mechanism

Implementations supporting `taskgraph` shall be able to generate a TDG, either at compile-time or at run-time, from a region annotated with the `taskgraph` clause. By leveraging the information contained in the TDG, the compiler or the runtime is capable of replacing the entire taskgraph region (meaning the user code) with the execution of the TDG. Therefore, not only the overhead related to task creation, dependency resolution and task deletion is alleviated, but also loop and conditional statements can be skipped.

Figure 2 illustrates the *taskgraph* mechanism. Particularly, Fig. 2a shows a snippet of a Heat transfer simulation implemented with OpenMP tasks and using the `taskgraph` clause, and Fig. 2b shows an overview of the TDG extracted from that application.

```
1   #pragma omp parallel
2   #pragma omp single
3   for (iter=0; iter < MAX_ITER; iter++)
4     #pragma omp task taskgraph {
5     for (i=0; i<M; i++) {
6       for (j=0; j<N; j++) {
7         #pragma omp task depend (in:Mat[i][j-1],
8                                  Mat[i][j+1],
9                                  Mat[i-1][j],
10                                 Mat[i+1][j])
11                         depend (out: Mat[i][j])
12        process_cell (i,j);
13      }
14    }
15  }
```

(a) Source code.



(b) Task Dependency Graph.

Fig. 2. Heat transfer simulation implemented with the Gauss-Seidel iterative method, using the proposed *taskgraph* clause

The main limitation of the *taskgraph* appears when the TDG has to be computed at run-time and, although the shape is stable for some time, there are conditions in the application that can make this TDG change. For those cases, the `taskgraph` clause can be declared with a list of variables, i.e., `taskgraph(`*`list`*`)` , which can be monitored at runtime, and so when a variable changes, the TDG is destroyed and rebuilt again. This mechanism is further described in Sect. 3.3.

3.2 Syntax of the `taskgraph` Clause

The proposed syntax for exposing a *taskgraph* region in OpenMP is as a new clause attached to the `task` or the `target` directives, as described next:

```
#pragma omp target|task [clause...] taskgraph [(list)]
```

With *clause* being the clauses currently allowed to be used with the corresponding directive, and *list* being the list of variables that shape the TDG, e.g., the loop boundaries if tasks are instantiated within a loop statement.

Although *taskgraph* allows a new execution model for OpenMP (named *define-once-run-repeatedly*, and described in the next subsection), there are some reasons leading us to define it as a clause instead of a new directive: (a) taskgraphs can be applied to both host and accelerator models, and so defining it as a directive would force to introduce additional clauses to describe where the taskgraph is to be executed; and (b) the *taskgraph* region can be seen as an implicit task with nested parallelism and, as such, it can benefit from clauses already defined for `task` directive like dependencies, priorities, and data-sharing clauses, or those defined specifically for `target` directive, like mapping clauses. Another option would be to add the new `taskgraph` clause to the `taskgroup` directive. However, this will remove the possibility of defining dependencies between the tasks in a taskgraph and previous/next tasks, and also reduce interoperability with the accelerator model. Overall, as of OpenMP 5.1 specification, there are 16 different types of constructs (that is, executable OpenMP directives, often attached to a block of user code), and 28 various constructs without counting the combined ones, each construct may have numerous associated clauses. By defining `taskgraph` as a clause to existing directives, we avoid rendering the specification more complex and we reduce the implementation effort it induces, because clauses as `depend` associated with tasks are currently implemented and can be directly used to build the TDG when `taskgraph` is declared.

3.3 Semantics of the `taskgraph` Clause

This section describes the semantics of the `taskgraph` clause in terms of the execution model and the memory model.

Execution Model. When a thread encounters a `task` or `target` directive declared with `taskgraph` clause, it will be exposed to one of the following situations: (a) there is missing information in the TDG of the taskgraph region, or (b) the TDG contains all task-related information in its structured-block, and its execution is equivalent (in terms of functionality) to the execution of the source code in the associated region. The procedure varies depending on the case:

- In the first case, the thread encountering the `taskgraph` clause executes the corresponding *taskgraph* region, and is also in charge of saving the missing information in the TDG runtime structures by, for instance, recording and saving the data captured during the execution of the inner tasks.

– In the second case, when the TDG is already complete, the encountering thread needs to launch its execution so that other idle threads can execute the TDG jointly. The user code in the `taskgraph` region will not drive the execution of the tasks, but the TDG instead.

Additionally, if `taskgraph` *`(list)`* is declared, the variables included in *list* shall be copied and saved when the region is executed for the first time. In other words, these are considered as *firstprivate* variables to the taskgraph region. While the program is running, the original copies of these variables in *list* can change. In this case, the update will be propagated to the TDG the next time we enter the `taskgraph` region, at which time the rebuild process of the TDG will start. The *list* is user-defined and shall include only variables defining the shape of the TDG, i.e., the variables defining the boundaries of loops or the branches taken in conditional statements enclosing the inner tasks, or the variables in the dependencies, if these change the memory object being dependent.

The TDG-driven execution can obtain its maximum efficiency when the *taskgraph* is defined once and replayed multiple times. This is the so-called *defined-once-run-repeatedly* execution model (as for CUDA graphs). Hence, implementations of `taskgraph` are recommended to build the TDG either at compile-time (if conditions allow, i.e., data size is known, loop boundaries are static, etc.) or after running the `taskgraph` region for the first time, at run-time, in order to maximize the performance gain of the subsequent executions.

The execution of the taskgraph region is synchronized by an implicit *taskgroup*. In other words, tasks created in the `taskgraph` structured-block belong to the same taskgroup set. The taskgroup is implicit and is declared as if it was surrounding the task defined by the directive combining with `taskgraph`.

Memory Model. The new `taskgraph` clause does not affect the existing OpenMP memory model regarding both the current global memory model, i.e. relaxed-consistency shared-memory model, and the interpretation of the data-sharing clauses that are attached to the task directives. However, the context generated by the `taskgraph` clause manages its data environment differently from how it is managed in a task.

More specifically, upon encountering a task directive (meaning `task` or `target`), all the clauses declared with the directive are immediately evaluated, including `taskgraph`. If `taskgraph` is executed, the declared data-sharing clauses also apply for the Task Dependency Graph. Inner-tasks may have different data-sharing clauses over the same data, e.g., a variable being global to the taskgraph can be set as private to tasks within it, using `firstprivate` or `private` clauses. Unlike `task` and `target`, where data environments are destroyed at completion, when a task accompanied by a `taskgraph` clause finishes its execution, all its data is recommended to be preserved, so the subsequent iterations can start without initialization. Programs that rely on saving the context of a `taskgraph` region after its completion to execute correctly are non-conforming and result in unspecified behavior.

3.4 Requirements of the `taskgraph` Region

A *taskgraph* region can be represented as a TDG, and so, it only stores information related to the execution of the inner tasks. As a result, the `taskgraph` clause is only applicable to those regions of code that are *completly taskified*, i.e., all the computation is done within the inner tasks, and the code in between only decides the control flow, so there cannot be sequential code in-between tasks.

While analyzing the `taskgraph` region, it can happen that the inner tasks contain nested tasks, as allowed in the current OpenMP specification. While syntactically correct, defining nested *taskgraph*s is however prohibited. In other words, *a* `taskgraph` *region can contain nested tasks, but none of them can be declared with* `taskgraph` *clause.* The reason is that taskgraph contains all information related to the execution of the tasks declared in its associated region, meaning that an inner taskgraph is entirely included in its outer taskgraph. Therefore, it is pointless to have nested taskgraphs, and it would break the semantics of the outer taskgraph if an inner taskgraph changes its shape in a different point in time than the outer one.

4 Projected Results

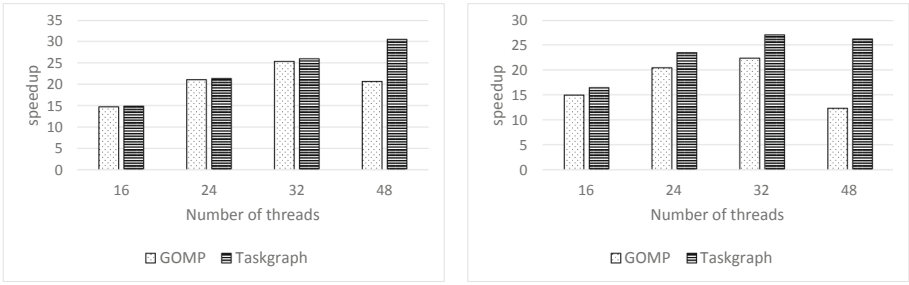
This section presents the expected results from integrating the `taskgraph` clause into the OpenMP specification. Two aspects are covered: (a) the potential performance gain from alleviating task management overhead and (b) the portability facilitated by the TDG to map OpenMP into other programming models.

4.1 Potential Performance Gain

The `taskgraph` clause targets the reduction of the overhead due to the orchestration of the parallel execution of tasks, comprising task creation, task enqueue and dependency resolution. According to Gautier et al. [3], who consider the LLVM libOMP runtime library, resolving task dependencies represents the major overhead source (up to 90%) when executing dependent tasks, and it further scales with the number of threads. In other words, using `taskgraph` can optimally relieve the greatest task overhead source and enhance the program scalability by alleviating the overhead related to multi-threading.

The results of our experiments support this statement, as shown in Fig. 3. In this example, we consider the heat transfer simulator and the HOG (Histogram of Oriented Gradients) object detection application, run on a Marenostrum cluster node, described in note 1. While the problem size and the task granularity are kept invariant, we modify the number of threads across the experiment. Both applications run for 128 iterations. Finally, the charts compare the execution using the original libgomp runtime library, labelled named *GOMP*, with the enhanced libgomp supporting the recording of the TDG at runtime, named *Taskgraph*. Particularly, *Taskgraph* version records the TDG in the first iteration and reuses it for the next 127 iterations. The figure shows that using the proposed *taskgraph* feature not only provides equivalent or better speedup than the

original libgomp in all considered scenarios, but it also allows the application to further benefit from the thread scaling.



(a) Speedup of HOG

(b) Speedup of Heat

Fig. 3. Speedup of Heat Transfer Simulation (using Gauss-Seidel method) and HOG object detection application, running 128 iterations, using original GOMP runtime library and a modified version with support for `taskgraph`

While the results seem promising, we must underline that it is preferable to use `taskgraph` for repeated task region (e.g., the computation loop inside simulators as N-body simulation or iterative problem solvers as the Gauss-Seidel method), because the first iteration will be charged by the generation of the TDG, incurring greater runtime overhead than the original runtime system. This is illustrated by Table 1, where we execute the kernels only once, with fixed number of threads (24 threads in this case, assigned to a single socket). The execution times are in milliseconds. GOMP execution corresponds to the time needed to execute the applications with the native GOMP runtime library. Similarly, Recorded execution is obtained with the modified library. The Record overhead is simply Record execution time minus the GOMP execution time. As the table depicts, the overhead incurred by the record mechanism increases when the task number increases. Another factor that may impact the cost of recording is the number of dependent variables, that is, the number of variables defining the dependency relationship among tasks. More specifically, the more dependent variables there are, the longer the dependency resolution will last, resulting in a longer record process.

Table 1. Time needed (millisecond) to execute the kernels once, with 24 threads

Application name	# tasks	GOMP execution	Recorded execution	**Record overhead**
Heat transfer	2560	20.3	23	**2.7**
	4000	19.8	23.9	**4.1**
HOG application	3600	48.5	52.1	**3.6**
	8040	46.9	54.4	**7.5**

4.2 The TDG: A Door for Expanding Portability

Task-based parallelism is very effective in uncovering the parallelism available in HPC applications. There are several programming models supporting tasking, e.g., OpenMP, Cilk++ [9], Intel TBB [7] and CUDA graphs [11] are among the more extended. The major success of OpenMP in front of its competitors substantiates in many factors: (a) it relies on relatively simple compile-time directives to expose parallelism (hence avoiding the need of refactoring sequential applications); (b) it is supported by a vast majority of compiler and chip vendors (including Intel, GCC and LLVM in the former, and Intel, ARM and PowerPC in the latter); and (c) it offers a great trade-off between programmability and performance, among others.

The Task Dependency Graph representing an OpenMP task-based application is however equivalent to that extracted when using other APIs to expose the parallelism. Figure 4 illustrates the portability enabled by means of the TDG. More specicifically, Fig. 4a shows a simple sequential code snippet, Fig. 4b shows the TDG representing the concurrency available in the sequential code, and Figs. 4c, 4d, 4e and 4f show the Cilk++, OpenMP, TBB and CUDA graph implementations of the TDG, respectively.

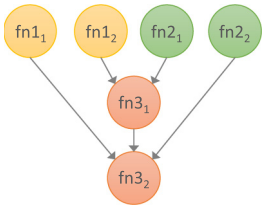
As the Figure depicts, OpenMP effectively offers better programmability by only introducing compile-time directives in an exact same version of the sequential code. Conversely, all Cilk++, TBB and CUDA graphs require some refactoring from the code for different reasons: (a) Cilk++ does not provide data-flow dependencies, but full synchronizations instead; (b) TBB decouples the description of the graph from its execution, and requires specific functions for starting the graph and joining results; and (c) CUDA graphs provide a low-level API that forces programmers to manage data copies and point-to-point synchronizations. The performance comparison between these models is out of the scope of this paper, but several works have already tackled this topic showing performance results for OpenMP competitive to the other parallel models [12, 17].

Previous works already studied the portability provided by the TDG to transform OpenMP task-based applications into CUDA graphs [20]. This approach uses the static computation of the TDG to lower the code into calls to the CUDA graph API instead of calls to a regular OpenMP implementation (e.g., GOMP or LLVM).

OmpSs is another example of interoperability based on the TDG. This programming model, developed by Barcelona Supercomputing Center, has been a forerunner of OpenMP with respect to the tasking model. Therefore, it supports tasked-based parallelism, and also heterogeneous computing with devices like GPUs and FPGAs [14]. The TDG extracted, at runtime, from the compile-time directives defined with OmpSs is used to manage tasks across heterogeneous architectures supporting different programming models like CUDA and OpenCL. Results show how OmpSs can fully replace the host API of both CUDA and OpenCL in a portable way.

```
int fn1(int v) {...}
int fn2(int v) {...}
int fn3(int v) {...}
int tmp1, tmp2, res=0;
for(int i=1; i<=2; ++i) {
  tmp1 = fn1(i);
  tmp2 = fn2(i);
  res += fn3(tmp1, tmp2);
}
```

(a) Serial version



(b) TDG

```
tmp1 = cilk_spawn fn1(1);
tmp2 = cilk_spawn fn2(1);
tmp3 = cilk_spawn fn1(2);
tmp4 = cilk_spawn fn2(2);
cilk_sync;
res += fn3(tmp1, tmp2);
res += fn3(tmp3, tmp4);
```

(c) Cilk++

```
#pragma omp parallel shared(res)
#pragma omp single
for(int i=1; i<=2; ++i) {
  #pragma omp task \
      depend(inoutset:tmp1) \
      shared(tmp1)
  tmp1 = fn1(i);
  #pragma omp task \
      depend(inoutset:tmp2) \
      shared(tmp1)
  tmp2 = fn2(i);
  #pragma omp task \
      depend(in:tmp1,tmp2) \
      depend(inout:res) \
      firstprivate(tmp1,tmp2) \
      shared(res)
  res += fn3(tmp1, tmp2);
```

(d) OpenMP

```
graph g;

broadcast_node<int> start(g);
function_node<int,int> n1(
     g, unlimited, fn1());
function_node<int,int> n2(
     g, unlimited, fn2());
join_node<tuple<int, int>,
     queueing> jn(g);
function_node<tuple<int,int>, int>
     n3(g, serial, fn3(res));

make_edge(start, n1);
make_edge(start, n2);
make_edge(n1, input_port<0>(jn));
make_edge(n2, input_port<1>(jn));
make_edge(jn, n3);

for(int i=1; i<=2; ++i)
     start.try_put(i);
g.wait_for_all();
```

(e) TBB

```
cudaGraph_t g;
cudaGraphCreate(&g, 0)

cudaGraphNode_t n1, n2, n3, n4, n5, n6;
cudaKernelNodeParams n1_args, n2_args, n3_args,
                     n4_args, n5_args, n6_args;
n1_args.func = (void*) fn1;
void *kernelArgs_n1[2] = {&i_1, &tmp1};
n1_args.kernelParams = (void**) kernelArgs_n1;
cudaGraphAddKernelNode(&n1, g, NULL, 0, &n1_args);

// similar to n1, but n2 uses fn2 and tmp2 instead
...
n3_args.func = (void*) fn1;
void *kernelArgs_n3[2] = {&i_2, &tmp3};
n3_args.kernelParams = (void**) kernelArgs_n3;
cudaGraphAddKernelNode(&n3, g, NULL, 0, &n3_args);
// similar to n3, but n4 uses fn2 and tmp4 instead
...
n5_args.func = (void*) fn3;
void *kernelArgs_n5[3] = {&tmp1, &tmp2, &res};
n5_args.kernelParams = (void**) kernelArgs_n5;
std::vector<cudaGraphNode_t> n5_deps;
n5_deps.push_back(n1); n5_deps.push_back(n2);
cudaGraphAddKernelNode(&n5, g, n5_deps.data(),
                       n5_deps.size(), &n5_args);
n6_args.func = (void*) fn3;
void *kernelArgs_n6[3] = {&tmp3, &tmp4, &res};
n6_args.kernelParams = (void**) kernelArgs_n6;
std::vector<cudaGraphNode_t> n6_deps;
n6_deps.push_back(n3); n6_deps.push_back(n4);
n6_deps.push_back(n5);
cudaGraphAddKernelNode(&n6, g, n6_deps.data(),
                       n6_deps.size(), &n6_args);

cudaGraphExec_t gExec;
cudaGraphInstantiate(&gExec, g, NULL, NULL, 0);
cudaStream_t gStream;
cudaStreamCreate(&gStream);
cudaGraphLaunch(gExec, gStream);
cudaStreamSynchronize(gStream);
cudaGraphExecDestroy(gExec);
cudaGraphDestroy(g);
cudaStreamDestroy(gStream);
```

(f) CUDA

Fig. 4. TDG representation and high-level description of a simple code parallelized with different task-based parallel programming models.

5 Related Work

The imminent advent of exascale computing raises new challenges in on-node parallelism, such as the efficient exploitation of modern many-core processors and the increasing heterogeneity of HPC systems. OpenMP is the current *de facto* standard parallel programming model, and it needs to address these challenges. Although the OpenMP tasking model is a convenient method to parallelize applications, many authors have investigated to tackle the overhead this model incurs [8,13,15]. That work is considered Sect. 1. This section focuses on work related to the Task Dependency Graph representation and its benefits.

M. Serrano et al. [16] provided a timing analysis over OpenMP tasks, where tasks with timing properties are represented in a TDG. This work strengthened the possibility of using OpenMP untied tasks (i.e., once such task is suspended by the initial thread, it can be correctly resumed by any idle thread within the same OpenMP team) on safety-critical embedded systems. A. Munera et al. [10] showed how statically generated TDGs can reduce the dynamic memory usage of OpenMP tasks, so that the tasking model can be used on embedded systems conveniently, where the amount of dynamic memory is often limited by safety constraints. Taskgraph makes OpenMP more suitable for safety-critical embedded systems by reinforcing their work:

- The `taskgraph` clause can be used with both `tied` and `untied` tasks, making the analysis of [16] still valid for taskgraph. As a result, taskgraph can perform the associated region in shorter time by reducing the runtime overhead, which eases the scheduling of the region within a larger real-time application.
- Techniques used in [10] rely on the static generation of the TDG, the information from which can be leveraged by the *taskgraph* clause to enhance the performance. Therefore, by including the new clause in their method, the resulting framework should deliver better performance than the current OpenMP tasking model, and also use less dynamic memory throughout the execution.

C. Yu et al. [20] proposed a framework to generate a CUDA graph [11] from OpenMP task directives. The new execution model proposed by Nvidia, where each node represents a CUDA kernel and edges express the dependencies among them, is interestingly similar to the taskgraph structure. This work shows, on the one hand, how OpenMP could benefit from a *define-once-run-repeatedly* execution model, as that enabled by CUDA graphs, in terms of performance. On the other hand, it shows how the programmability of CUDA graphs could be enhanced by reducing the number of lines required from the programmer, going from 15500 to 4 in a Cholesky implementation used for illustration purposes.

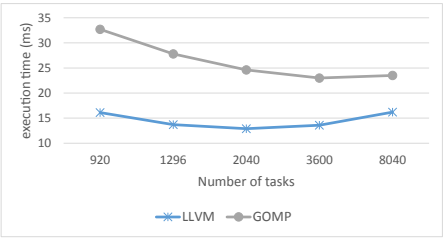
6 Conclusion

This work describes a new method to tackle the OpenMP task overhead at a higher abstraction level, that is, by introducing the concept of Task Dependency Graph in the OpenMP specification through `taskgraph` clause.

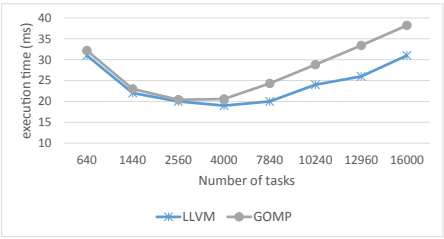
Our preliminary results, based on GCC GOMP runtime library, validate the effectiveness of the TDG, as a representation of a region of code that can be boosted by the OpenMP framework. When the TDG holds the complete execution of a part of the user's code, this code can be replaced by the execution of the TDG. This results in the reduction of the overhead introduced by the access to shared resources, like task queues, and the management of tasks, including creation, orchestration and destruction.

The concept of TDG also allows a new execution model in OpenMP, the *define-once-run-repeatedly* model, equivalent to that described by CUDA graphs. This mechanism, which is a hint for the implementation and shall not change the functional behavior of the program, allows further alleviating the overhead in applications running several times the same TDGs. Interestingly, this proposed mechanism can promote the use of the OpenMP API as a door for effectively exploiting CUDA graphs.

Future investigations include implementing the `taskgraph` clause in major compilers and runtime systems, such as LLVM, to further validate our results. As a prediction, we expect `taskgraph` to deliver significant performance gain in LLVM, as in GOMP library. This assumption is supported by Fig. 5, where we run different applications with the OpenMP runtime libraries from LLVM and GCC. Although the LLVM is better optimized in these cases (shorter execution time), its runtime overhead increases when the task granularity shrinks, similar to the GOMP library. Other research lines comprise (a) thoroughly testing the performance impact of the new clause in larger applications and different processor architectures; (b) using the taskgraph in applications with tasks inside a taskgraph region and tasks outside the region; and (c) exploring usages and improvements of other programming models through the use of the TDG generated by OpenMP `taskgraph`.



(a) HOG object detection application



(b) Heat transfer simulator

Fig. 5. Execution time (in ms) analysis of different applications with original GCC GOMP library and LLVM OMP runtime library, fixing the number of threads to 24

Acknowledgements. This work has been supported by the EU H2020 project AMPERE under the grant agreement no. 871669.

References

1. BSC: Marenostrum IV User's Guide (2017). https://www.bsc.es/support/MareNostrum4-ug.pdf
2. Castello, A., Seo, S., Mayo, R., Balaji, P., Quintana-Orti, E.S., Pena, A.J.: GLTO: on the adequacy of lightweight thread approaches for openmp implementations. In: Proceedings of the International Conference on Parallel Processing, pp. 60–69 (2017)
3. Gautier, T., Perez, C., Richard, J.: On the impact of OpenMP task granularity. In: de Supinski, B.R., Valero-Lara, P., Martorell, X., Mateo Bellido, S., Labarta, J. (eds.) IWOMP 2018. LNCS, vol. 11128, pp. 205–221. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98521-3_14
4. Giannozzi, P., et al.: Quantum espresso: a modular and open-source software project for quantum simulations of materials. J. Phys. Condens. Matter **21**(39), 395502 (2009)
5. Kalray MPPA products (2021). https://www.kalrayinc.com/
6. Komatitsch, D., Tromp, J.: SPECFEM3D Cartesian (2021). https://github.com/geodynamics/specfem3d
7. Kukanov, A., Voss, M.J.: The foundations for scalable multi-core software in intel threading building blocks. Intel Technol. J. **11**(4), 309–322 (2007). http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=79B311F4CEB9A4B610520177C7144D57?doi=10.1.1.71.8289&rep=rep1&type=pdf
8. Lagrone, J., Aribuki, A., Chapman, B.: A set of microbenchmarks for measuring OpenMP task overheads. In: Proceedingis of International Conference on Parallel and Distributed Processing Techniques and Applications II, pp. 594–600 (2011). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.217.9615&rep=rep1&type=pdf
9. Leiserson, C.E.: The Cilk++ concurrency platform. J. Supercomput. **51**(3), 244–257 (2010)
10. Munera, A., Royuela, S., Quinones, E.: Towards a qualifiable OpenMP framework for embedded systems. In: Proceedings of the 2020 Design, Automation and Test in Europe Conference and Exhibition, DATE 2020, no. 2, pp. 903–908 (2020)
11. Nvidia: CUDA Graph programming guide (2021). https://docs.nvidia.com/cuda/cuda-c-programming-guide/#cuda-graphs
12. Olivier, S.L., Prins, J.F.: Evaluating OpenMP 3.0 run time systems on unbalanced task graphs. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 63–78. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02303-3_6
13. Perez, J.M., Beltran, V., Labarta, J., Ayguade, E.: Improving the integration of task nesting and dependencies in OpenMP. In: Proceedings - 2017 IEEE 31st International Parallel and Distributed Processing Symposium, IPDPS 2017, pp. 809–818 (2017)
14. Sainz, F., Mateo, S., Beltran, V., Bosque, J.L., Martorell, X., Ayguadé, E.: Leveraging OmpSs to exploit hardware accelerators. In: 2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing, pp. 112–119. IEEE (2014)

15. Schuchart, J., Nachtmann, M., Gracia, J.: Patterns for OpenMP task data dependency overhead measurements. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) Scaling OpenMP for Exascale Performance and Portability, pp. 156–168. Springer International Publishing, Cham (2017)
16. Serrano, M.A., Melani, A., Vargas, R., Marongiu, A., Bertogna, M., Quiñones, E.: Timing characterization of OpenMP4 tasking model. In: 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES 2015, pp. 157–166 (2015)
17. Stpiczyński, P.: Language-based vectorization and parallelization using intrinsics, openmp, tbb and cilk plus. J. Supercomput. **74**(4), 1461–1472 (2018)
18. TOP500 (2020). https://www.top500.org/lists/top500/2020/11/
19. Valero-Lara, P., Catalán, S., Martorell, X., Usui, T., Labarta, J.: sLASs: a fully automatic auto-tuned linear algebra library based on openmp extensions implemented in ompss (lass library). J. Parallel Distrib. Comput. **138**, 153–171 (2020)
20. Yu, C., Royuela, S., Quiñones, E.: OpenMP to CUDA graphs: a compiler-based transformation to enhance the programmability of NVIDIA devices. In: Proceedings of the 23rd International Workshop on Software and Compilers for Embedded Systems, SCOPES 2020, pp. 42–47 (2020)