OpenMP taskloop dependences

Marcos Maroñas[1,2], Xavier Teruel[1], and Vicenç Beltran[1]

[1] Barcelona Supercomputing Center (BSC), Spain
{mmaronas, xteruel, vbeltran}@bsc.es
http://www.bsc.es
[2] Huawei Research, Edinburgh
marcos.maronas.bravo@huawei.com

Abstract. Exascale systems will contain multicore/manycore processors with high core count in each node. Therefore, using a model that relaxes the synchronization, such as data-flow, is crucial to adequately exploit the potential of the hardware. The flexibility of the data-flow execution model relies on the dynamic management of data-dependences among tasks.
The OpenMP standard already provides a construct, known as `taskloop`, that distributes the loop iteration space into several tasks, but this construct does not support the use of the `depend` clause yet. In this paper we propose the use of the induction variable to define data dependences in tasks created by the `taskloop` construct. By using the induction variable, each task will contain its own dependences based on the partition of work they received.
We also aim to demonstrate that using taskloop with dependences provides an enhancement in terms of programmability with respect to using stand-alone tasks to parallelize a loop. Our implementation does not introduce any significant overhead on the taskloop implementation and, in certain cases, it outperforms the stand-alone task version.

Keywords: OpenMP · Tasking · Loops · Synchronization · taskloop construct · depend clause.

1 Introduction

The introduction of the first multiprocessor architectures led to the development of shared memory programming models. One of those is OpenMP, which became a *de facto* standard for parallelization on shared memory environments.

OpenMP [12], with its highly optimized fork-join execution model, is a good choice to exploit structured parallelism, especially when the number of cores is small. Worksharing constructs, like the well-known `omp for` construct, are good examples of how OpenMP can efficiently exploit structured parallelism. However, when the number of cores increase and the work distribution is not perfectly balanced, the rigid fork-join execution model can hinder performance.

The `omp for` construct accepts different scheduling policies that can mitigate load-balancing issues; and the `nowait` clause avoids the implicit barrier at the

end of an `omp for`. Still, both techniques are only effective in a few specific situations. Moreover, the fork-join execution model is not well-suited for exploiting irregular, dynamic, or nested parallelism.

Task-based programming models were developed to overcome some of the above-mentioned limitations. The first tasking models were based solely on the tasks and taskwaits primitives, which naturally support irregular, dynamic, and nested parallelism. However, these tasking models are still based on the fork-join execution model. The big step forward came with the introduction of data dependences. Thus, replacing the rigid fork-join execution model by a more flexible data-flow execution model that relies on fine-grained synchronizations among tasks. Modern task-based programming models such as as Cilk, OmpSs or OpenMP tasking model have evolved with advanced features to exploit nested parallelism [13], hardware accelerators [2][6][1], and seamless integration with message passing APIs such as MPI [15][14].

Exascale systems will contain multicore/manycore processors with high core count in each node. Therefore, using a model that relaxes the synchronization, such as data-flow, is crucial to adequately exploit the potential of the hardware.

Additionally, worksharing techniques are easier to apply compared to tasking. A single worksharing construct is enough to parallelize a loop. In contrast, using tasks, it requires more effort from the user. There must be at least a task per core, to feed all the cores and prevent lack of parallelism. A frequent technique applied to create enough tasks is blocking. This technique partitions a loop in several blocks, and each block is processed by an independent task. Although this is not a complex technique, it requires more effort than the worksharing alternative.

The OpenMP standard contains a directive able to distribute the iteration space of a loop into tasks, which, theoretically, enables users to parallelize a whole loop with a single construct using tasks. This is the `taskloop` construct. However, in practice, it is not useful for a single reason: it does not support data dependences. Thus, a `taskloop` creates a set of tasks that cannot have data dependences, and so, the synchronization must be done using coarse-grained synchronization points (i.e., the implicit taskgroup, or explicit taskwaits). So, basically, we end up in a fork-join model but with increased overhead compared to worksharing constructs.

We propose adding support for data dependences to the `taskloop` construct. Our proposal enables programmers to use the induction variable of the loop to specify data dependences. Thus, each task created by the taskloop will register the data dependences specified by the user. If the induction variable is used to specify any dependence, each task will register the dependence using its own value of the induction variable. As a result, apply blocking is possible using a single construct, enhancing programmability.

2 Tasking Programmability Challenges

Compared to using worksharing techniques, tasks are more complicated to use. If we simply replace worksharing constructs by task constructs, there is very few parallelism, and most of the cores are idle. This is because a worksharing construct distributes the work among all the available cores, that run concurrently. In contrast, an instantiated task is a piece of code that runs only in a single core at a given instant of time. Figure 1 shows such a problem. The figure also shows one possible solution, which is the use of blocking.

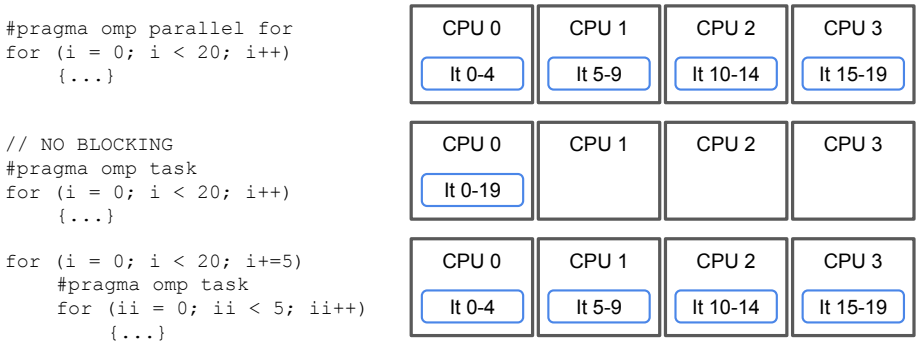

Fig. 1. Illustration of CPU occupation using different parallelism techniques.

Listings 1.1 and 1.2 shows a real code using worksharing constructs and task with blocking respectively. It is possible to see that applying blocking techniques is simple, but also that it requires more effort than using worksharing constructs. For a single loop, worksharing constructs require only three lines of code, while tasks with blockings require five lines of code.

The OpenMP standard already provides a construct known as `taskloop` that distributes work into several tasks. This construct is the natural replacement of worksharing constructs to use tasks. Listing 1.3 shows the very same example using the `taskloop` construct. Notwithstanding, the tasks created using the `taskloop` construct cannot have data dependences, so they can only be synchronized using synchronization points (i.e., keeping the implicit taskgroup region or using explicit taskwaits). As a result, we have a fork-join pattern with its rigid synchronization. So, we moved from worksharing constructs to tasks to benefit from a more lightweight data-flow synchronization, but the impossibility of using data dependences when using the `taskloop` construct prevents us from obtaining all its benefits.

In summary, tasks require more effort from users than worksharing constructs. However, tasks provide key benefits that fit the requirements of Exascale systems better than worksharing constructs. For instance, programmers

Listing 1.1. Simple code using the work-sharing construct

```
#pragma omp for
for(size_t j = 0; j < N; j++)
    b[j] = scalar*c[j];
```

Listing 1.2. Simple code using the task construct (and blocking)

```
for(size_t j = 0; j < N; j+=BS) {
    size_t size = j+BSIZE > N ? N-j : BSIZE;
    #pragma omp task depend(in:c[j:size]) depend(out:b[j:size])
    for(size_t j2=j; j2 < j+size; j2++)
        b[j2] = scalar*c[j2];
}
```

Listing 1.3. Simple code using the taskloop construct

```
#pragma omp taskloop chunksize(BSIZE)
for(size_t j = 0; j < N; j++) {
    b[j] = scalar*c[j];
} // implicit taskgroup region
```

may parallelize different stages of the program by means of tasks, as long as they guarantee a proper task depend annotation. Then, they can rely on the OpenMP run-time library to compute the correct synchronization order among these tasks. In other words, task parallelization improves composability.

In the other hand, the `taskloop` construct enables programmers to use tasks with a similar effort than the effort required by worksharing constructs. Nevertheless, it does not support data dependences, and this prevent users from getting the key benefits of tasking.

3 Related work

We already mentioned in the previous section that OpenMP supports both loop-based parallelism and task-based parallelism. The most common way of using loop-based parallelism in OpenMP is by means of the worksharing constructs. In terms of programmability, worksharing constructs enable users to parallelize loops using a single construct. Thus, they are very simple to use. In terms of performance, worksharing constructs deliver good performance in the general case. Nevertheless, they contain an implicit barrier at the end of the worksharing region, introducing very rigid synchronization.

The task-based approach is a bit more complex in terms of programmability. It usually requires blocking techniques to uncover parallelism, which require some more code than a single construct. Regarding performance, tasks have a natural ability to deal with load imbalance, but they have associated costs that may introduce some overhead depending on task's granularity. OpenMP pro-

vides also the `taskloop` construct, that distributes the iteration space of a loop into several tasks. There is the possibility of specifying a *grainsize* guaranteeing that each of the tasks created executes no less than *grainsize* iterations. Thus, the taskloop construct simplifies the use of task-based parallelism, enabling users to parallelize loops with a single construct. Nevertheless, OpenMP does not support dependences in the taskloop construct. As a result, users must rely on fork-join-like synchronization with explicit synchronization points. Consequently, dropping the data-flow execution model of task-based parallelism, and its benefits. By enabling the use of data dependences in the taskloop construct, we offer users the possibility of parallelizing loops in a single construct while keeping the benefits of the data-flow execution model. Additionally, the use of the taskloop construct, may reduce the tasking overhead because allocations could be optimized to be done as a whole, instead of one by one. However, the number of tasks that will be created and scheduled is still proportional to the problem size.

On the other hand, OpenMP 4.5 [11] already included the `doacross` dependences for work-sharing loops based on the *source* and *sink* dependence types and the extension of the `ordered` construct to support the `depend` clause. This feature allows to express general cross-iteration loop dependences and thereby support doacross parallelization [16]. The loop dependencies that cross the iteration space are enforced via point-to-point synchronization injected where the compiler finds the `ordered` construct. The main advantage of this extension is that only applying to a unique iteration space, it could be easily implemented by a single 2D matrix of *per chunk* relationships, imposing a very low overhead to the runtime. This advantage is also its main drawback, as it does not allow to dynamically connect to other loop iteration spaces (neither combine it with any other work-sharing or task generating constructs).

Intel Cilk presents the `cilk_for` [7], which is used to parallelize loops. The body of the loop is converted into a function that is called recursively using a divide and conquer strategy for achieving better performance. However, there is a `cilk_sync` at the end of each iteration. Therefore, synchronization is quite rigid, similarly to OpenMP worksharings. Moreover, Cilk tasks do not support data dependencies between tasks.

The CUDA programming model [10, 8] allows expressing kernel dependencies using proper streams and events. This approach will be similar to the stand-alone task model implemented by the OpenMP standard and requires manually transform the loop to its blocked version, to decompose the iteration space. Furthermore, the use of multiple streams and their synchronization via events could be non-trivial in some cases. On the other hand, CUDA offers the option to capture and reuse such task instantiation using the CUDA graph set of routines, offering an extra optimization by reducing the overhead. The CUDA graph functionality is out of the scope of this paper and should be considered as an extension that can be widely used, not only in taskloop constructs but in any portion of code parallelized with tasks.

```
for (t = 0; t < 100; t++)
  #pragma omp taskloop grainsize(5) \\
   depend(inout: x[i])
  for (i = 0; i < 20; i++)
    {...}
```

```
for (t = 0; t < 100; t++)
  for (i = 0; i < 20; i+=5)
    #pragma omp task depend(inout:x[i])
     for (ii = i; ii < i+5; ii++)
       {...}
```

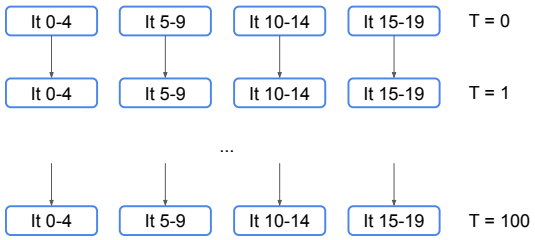

Fig. 2. Partition of work and dependences between tasks created using taskloop

4 Taskloop with Dependences

In this section, we detail the syntax of our proposal to support data dependences in the `taskloop` construct. In short, we propose the use of the induction variable to define data dependences in tasks created by the `taskloop` construct. By using the induction variable, each task will contain its own dependences based on the partition of work they received. Figure 2 shows an example. There is an outer loop and an inner loop parallelized using the enhanced `taskloop` construct with data dependences. As we can see, the data dependences contain the induction variable. In this case, it means that each of the tasks register a dependence over the `i`-th element of `x`. As each of the tasks receive a part of the iteration space, each of the tasks will have different values for `i`, thereby allowing them to run concurrently, but defining a dependence with tasks of the next and previous `t` iteration that work over the same data. Figure 2 also includes the code to get equivalent behavior using regular tasks.

With this mechanism, expressivity is enhanced and the `taskloop` construct becomes usable in many real-world examples while keeping the key benefits of tasking.

We would like to point out that the mechanism to define the granularity of a task created by the `taskloop` construct is the `grainsize` clause, as shown in Figure 2. As the loop iteration distribution should be deterministic and visible to the programmer (in order to allow him/her to combine tasks generated by the taskloop with stand-alone generated tasks), the use of taskloop dependences will assume the strict modifier within the `grainsize` clause. If no `grainsize` is provided, default applies.

A second observation relates the task instantiation order and the dependence computation. Currently, taskloop does not guarantee any task creation order, while the proper dependence computation relies on it. Although the implementation does not need to actually create tasks in such order, it must compute dependencies AS IF they were created in the logical iteration order. The OpenMP

runtime can easily follow this behavior due to it can compute the number of tasks and the corresponding task boundaries when encountering the taskloop construct. In addition, the current specification says *"Programs that rely on any execution order of the logical iteration are non-conforming"*; such part of the specification must be relaxed to take into account synchronization arising from dependences.

Although in our implementation the compiler is able to detect the use of the loop control variable and modify the corresponding value for each task instance, a more OpenMP-aligned approach would involve the use of a modifier within the `depend` clause. We may have two different options:

- Leverage the existent iterator modifier by extending its syntax to accept a new loop-based value generator. Then, the example in Figure 2 could be rewritten by annotating the loop using:
  `#pragma omp taskloop depend(iterator(t=taskrange), inout: x[t]) grainsize(5)`.

- Define a new modifier based on the task loop boundaries. Again, the example in Figure 2 could be rewritten using:
  `#pragma omp taskloop depend(taskrange(b=begin), inout: x[b:5]) grainsize(5)`.

While the first option relies in the multiple values generated by the iterator modifer (by means of a new iterator range specifier: *taskrange*); the second one uses the chunk-associated lower bound (by means of the new OpenMP keyword `begin` and a new depend modifier: *taskrange*) combined with the grainsize parameter in order to define an array section.

Finally, adding taskloop dependences on the OpenMP specification should relax the implicit taskgroup region defined as part of the `taskloop` construct. It will be recommended to implicitely consider the `nogroup` clause when the programmer uses any dependence clause on the `taskloop` construct.

5 Implementation

Our proposal is done in the OmpSs-2 programming model, built on top of the Mercurium compiler and the Nanos6 runtime library. Following, we detail the extensions done in both components to support dependences in the `taskloop` construct. We also conceptually explain our implementation.

Semantics The `taskloop` construct is a convenient "syntactic sugar" to ease the use of tasks. It can be implemented just by applying an automatic blocking technique in the compiler side, similar to a manual blocking done by the end-user. But it also allows smarter implementations that can improve performance by reducing the associated overhead.

Mercurium compiler The Mercurium compiler has been extended to support the use of data dependences in the `taskloop` construct. Mercurium is a source to source compiler, meaning that it receives code as an input, and generates code as an output. Mercurium creates a function to register dependences per each task construct found in the user code. To support the use of the induction variable in the `taskloop` dependences, Mercurium has to accept a new parameter in the functions used to register dependences. Given that in our implementation it is the runtime who partitions the work and assigns iterations to the tasks, Mercurium must receive the information of the assigned iterations to replace the induction variable by its real value.

Additionally, in the same line, Mercurium creates a function per task type including the user code that the task has to run. In this case, it also has to receive an additional parameter: the iterations that each task has to run.

Finally, when creating a `taskloop` entity, Mercurium has to enable some flags to let the runtime system know that this is not a regular task, but a `taskloop`.

Nanos6 runtime library In the runtime system, the first step is to extend the work descriptor of a task to include the iteration space of the loop, and the grainsize specified by the user, if any. This single task will represent the whole taskloop and it will register the whole set of dependencies, which are generated based on the iteration space and the grainsize value. When the task instance that represents the taskloop is executed it will instantiate one sub-task for each partition of the iteration space with its corresponding dependencies. This approach works because we are leveraging the *weak dependencies* and *early release* features of OmpSs-2 [13] that enables the parent task to become ready even if the dependencies are still not fulfilled. In this way, the sub-tasks created behave as if they were created on the dependency domain of the parent taskloop. A side effect of this implementation is that the creation and execution of the taskloop is not blocking, so many of them can be executed in parallel. Each sub-task created inside a task-loop will behave as a regular task, waiting for its data-dependencies before it can become ready.

We would like to point out that our current implementation focuses on programmability. Therefore, we are trying to provide a simpler way of using tasks that introduces no significant overhead compared to using other techniques such as manual blocking. Nevertheless, the `taskloop` construct provides the opportunity to apply further optimizations that cannot be applied in the case of manual blocking. Such optimizations could include a single allocation for all the loop tasks, instead of allocating space for each of them individually; or the application of smarter throttle policies to mitigate memory overuse when there are too many tasks in flight. In this way, throttle policies may take into account the total number of tasks needed to instantiate the whole loop iteration space, rather than consider each of the tasks as an independent entity (i.e., making decisions based on each individual item).

6 Experiment results

In this section, we wish to demonstrate that the taskloop with dependences provides an enhancement in programmability when using tasks, while introducing no significant overhead compared to a manual equivalent implementation. For that purpose, our evaluation will focus in both programmability and performance.

Regarding programmability, we used several different metrics to compare the different implementations: Source Lines of Code (SLOC) [9], Development Estimate Effort (DEE) [5], and Cyclomatic Complexity (CC) [17]. It is important to highlight that for the SLOC metric we only consider the code related to the parallelization. And it is also important to notice that this metric is an approximation to measure code complexity based only on the number of lines but it ignores that some individual lines can be more complex than others.

In terms of performance, we compare the different implementations to demonstrate that using the `taskloop` construct do not add any significant overhead.

Environment The experiments were carried out on the Marenostrum 4 supercomputer. It is composed of nodes with 2 sockets Intel Xeon Platinum 8160 2.1GHz 24-core and 96GB of main memory.

Regarding the software, we used the Mercurium [3] compiler (v2.3.0) and the Nanos6 runtime library [4] as the baseline components to implement our proposal (described in Section 5); the gcc and gfortran compilers (v7.2.0), in order to compile Mercurium and Nanos6 (included here for reproducibility purposes); and the Intel compilers (v17.0.4), as the native compiler used by Mercurium to generate binary code.

Methodology As previously introduced, we focus our evaluation in two different aspects: performance and programmability. Our experiments will use two different versions of each application/benchmark:

- **T**. Version using regular tasks. It requires manual blocking.
- **TL**. Version using the `taskloop` construct with dependences.

Regarding performance, for each of the benchmarks/applications, we select two different problem sizes, one small-medium size, and one big size. For each of the problem sizes, we try several block sizes to show that the differences between the `T` and the `TL` are small or even non-existent in several different scenarios.

All the experiments ran using the interleaving policy offered by the numactl command, spreading the data evenly across all the available NUMA nodes, in order to minimize the NUMA effect.

The results shown in the figures are averages of five different executions. We decided to use only five executions because the variability across different executions was very small.

Related to programmability, we count the SLOC required to parallelize the baseline code for each of the versions, and use the SLOCCount [18] tool and the Lizard [19] tool to retrieve the DEE and CC respectively.

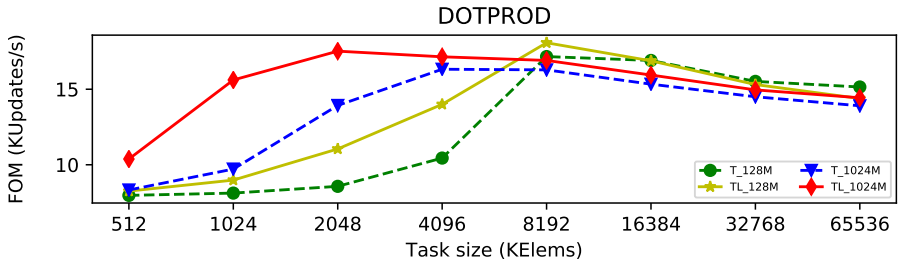

Fig. 3. Evaluation of taskloop using DOTPROD benchmark

Performance Evaluation In this section, we evaluate several applications/benchmarks to demonstrate that the use of the `taskloop` construct does not introduce overhead compared to a manual alternative. All the figures show the Figure of Merit (FOM) of the application on the y-axis, and different task granularities in the x-axis. All of them have four series: one using the `T` version with a small-medium problem size, one using the `TL` version with a small-medium problem size, one using the `T` version with a big problem size, and one using the `TL` version with a big problem size. We would like to highlight that the `T` versions use the usual approach where a single core creates all the tasks.

Figure 3 shows the results of the dot product benchmark. In this case, we repeat the *dotprod* kernel a given number of iterations to make the execution longer. For both problem sizes, the `TL` version performs better than the `T` version in the small task sizes. The `T` version has only a single core creating tasks. When the granularity is small, a single creator cannot create rapidly enough to feed all the cores. As explained in section 5, the `TL` version may have several cores creating tasks, speeding up the creation, and increasing the overall performance. The `TL` version may have several cores producing tasks because each iteration of the kernel is a taskloop instance, that can be running in different cores concurrently, while the `T` version has a single core creating all the iterations sequentially. Finally, from TS=8192, all the versions perform very similarly.

Figure 4 shows the results of the N-body benchmark. For this benchmark, we see again a difference in the smallest granularity, where the `TL` version outperform the `T` version for both problem sizes. Like previously, this is because there are several taskloops that can be creating tasks concurrently in the `TL` version, while there is a single core creating tasks sequentially in the `T` version, and it is not quick enough to feed all the cores.

Finally, Figure 5 shows the results of the Stream benchmark. The results presented are an average of the four different kernels of the Stream benchmark. In this benchmark, there are some differences between the `T` and `TL` versions. Firstly, in the smallest granularity, the `TL` version outperforms the `T` version in both problem sizes. Like in some previous benchmarks, this is because the `TL` version has multiple taskloops that can create concurrently rather than a single one, and speeding up the creation improves the overall performance. Then, when TS=64,

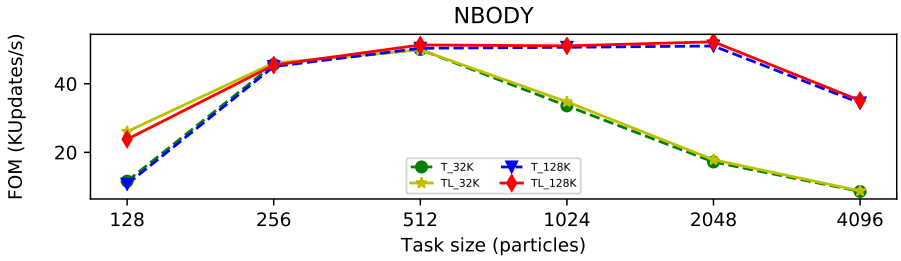

Fig. 4. Evaluation of taskloop using NBODY benchmark

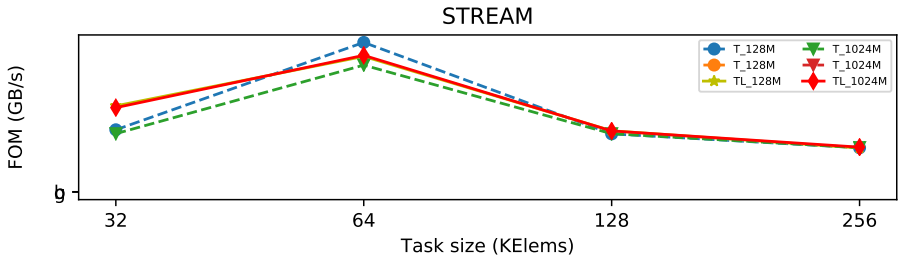

Fig. 5. Evaluation of taskloop using STREAM benchmark

for the small problem size the `T` version outperforms the `TL` version, and the other way around for the big problem size. Our runtime system has an immediate successor mechanism to exploit data locality between successor tasks. In this case, this mechanism is making the difference. We repeated the experiment with no immediate successor, and the results for both versions were very similar. For the big problem size, the `TL` version is able to find more immediate successor tasks than the `T` version, and the other way around for the small problem size.

Overall, we see that there are few differences between the `T` and `TL` versions, with the `TL` versions generally outperforming the `T` versions in fine granularities, thanks to the use of multiple creators. Thus, we can conclude that the `TL` is not only introducing very few overhead, but it is able to enhance performance in some specific scenarios.

Programmability Evaluation Table 1 shows the different programmability metrics evaluated in this analysis for different benchmarks. The DEE is a metric that tries to estimate the effort that a developer must spend to write a given code. In this case, it is measured in *person-months*. The size of the code affects the DEE. The CC metric is higher when a code can take more different paths. For instance, adding an `if` increases the CC.

In Table 1, some benchmarks show no difference between the `T` version and the `TL` version. As previously explained, a frequent way of parallelize an applica-

Listing 1.4. Stream code using tasks with blocking

```
for (int k=0; k<nTimes; k++) {
    for(size_t block = 0; block < NUM_BLOCKS; block++) {
        size_t aux = block*BSIZE;
        size_t size = aux+BSIZE > N ? N-aux : BSIZE;
        #pragma oss task in(a[aux;size]) out(c[aux;size])
        for(size_t j2=aux; j2 < aux+size; j2++)
            c[j2] = a[j2];
    }
    for(size_t block = 0; block < NUM_BLOCKS; block++) {
        size_t aux = block*BSIZE;
        size_t size = aux+BSIZE > N ? N-aux : BSIZE;
        #pragma oss task in(c[aux;size]) out(b[aux;size])
        for(size_t j2=aux; j2 < aux+size; j2++)
            b[j2] = scalar*c[j2];
    }
    for(size_t block = 0; block < NUM_BLOCKS; block++) {
        size_t aux = block*BSIZE;
        size_t size = aux+BSIZE > N ? N-aux : BSIZE;
        #pragma oss task in(a[aux;size], b[aux;size]) out(c[aux;size])
        for(size_t j2=aux; j2 < aux+size; j2++)
            c[j2] = a[j2]+b[j2];
    }
    for(size_t block = 0; block < NUM_BLOCKS; block++) {
        size_t aux = block*BSIZE;
        size_t size = aux+BSIZE > N ? N-aux : BSIZE;
        #pragma oss task in(b[aux;size], c[aux;size]) out(a[aux;size])
        for(size_t j2=aux; j2 < aux+size; j2++)
            a[j2] = b[j2]+scalar*c[j2];
    }
}
#pragma oss taskwait
```

tion with tasks is using blocking, thereby converting a single loop into two loops, one to iterate over blocks, and one to iterate over the elements of each block. The

Table 1. Programmability metrics to compare the use of the `taskloop` construct with the manual use of tasks

	T			TL		
	SLOC	**DEE**	**CC**	**SLOC**	**DEE**	**CC**
DOTPROD	7	0.07	4	3	0.05	2.5
NBODY	10	0.29	2.6	10	0.29	2.6
STREAM	25	0.37	13.5	7	0.3	9.5

Listing 1.5. Stream code using taskloop

```
for (int k=0; k<nTimes; k++) {
    #pragma oss taskloop grainsize(BSIZE) in(a[j]) out(c[j])
    for(j = 0; j < N; j++) {
        c[j] = a[j];
    }

    #pragma oss taskloop grainsize(BSIZE) in(c[j]) out(b[j])
    for(j = 0; j < N; j++) {
        b[j] = scalar*c[j];
    }

    #pragma oss taskloop grainsize(BSIZE) in(a[j], b[j]) out(c[j])
    for(j = 0; j < N; j++) {
        c[j] = a[j]+b[j];
    }

    #pragma oss taskloop grainsize(BSIZE) in(b[j], c[j]) out(a[j])
    for(j = 0; j < N; j++) {
        a[j] = b[j]+scalar*c[j];
    }
}
#pragma oss taskwait
```

`taskloop` construct prevents users from requiring this twofold loop structure in some cases, saving some lines of code. In the case of the Nbody benchmark, the data layout in our implementation is blocked, so we cannot eliminate the twofold loop. Thus, there is no real improvement in programmability for this benchmarks. In contrast, the Stream benchmark shows improvements in all the different metrics, reaching up to a 3.57x reduction of SLOC. Regarding the CC, using taskloop reduces it from 13.5 to 9.5 because we are able to remove four different `for` loops, as you can see comparing the code snippets in Listings 1.4 and 1.5. Similarly, in the dotprod benchmark we can remove two `for` loops, reducing the CC.

7 Conclusions and future work

The `taskloop` construct is a directive that distributes the iteration space of a loop into several tasks. This gives a boost to productivity when using the tasking model. However, the current implementation of this construct does not cover the vast majority of cases, because it is missing data dependences support. Therefore, users are forced to use explicit synchronization points, like in the fork-join model. As a result, users get a fork-join like structure, with increased overhead compared to worksharing constructs.

By providing support for data dependences to the `taskloop` construct, we enable users to utilize this directive in the majority of cases. Thus, they are able to fully convert their loops into tasks with a single directive, maximizing productivity, while keeping the main benefit of tasks, a lightweight synchronization based on data dependences.

Our evaluation shows that `taskloop` with dependences delivers as much performance as its manual counterpart, but with a reduced number of lines of code. The number of lines of source code required to parallelize a code using `taskloop` with dependences is up to 3.57x times smaller than its manual counterpart.

As future work, we plan to further investigate the interactions of `taskloop` dependences with other OpenMP features. The big challenge will imply combining dependences with the `collapse` clause. As this clause specifies the number of loops that are collapsed into a logical iteration space that then is divided according to the `grainsize` and `num_tasks` clauses; the implementation should take into account partial innermost level loop partition that will hinder the array section definition.

Another opportunity for future research include the study of the interaction with the loop transformation constructs (i.e., the `tile` and the `unroll` constructs). Although it seems the programmer should apply the semantics of the `taskloop` dependences on top of the already transformed loops, such use cases should be considered to discard potential corner cases.

Acknowledgements

This research has received funding from the European Union's Horizon 2020/EuroHPC research and innovation programme under grant agreement No 955606 (DEEP-SEA); and the support of the Spanish Ministry of Science and Innovation (Computacion de Altas Prestaciones VIII: PID2019-107255GB).

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2), 187–198 (2011)
2. Ayguade, E., Badia, R.M., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., et al.: A proposal to extend the OpenMP tasking model for heterogeneous architectures. In: International Workshop on OpenMP. pp. 154–167. Springer (2009)
3. Barcelona Supercomputing Center: Mercurium Compiler, https://github.com/bsc-pm/mcxx, accessed: 2019-03-24
4. Barcelona Supercomputing Center: Nanos6 Runtime, https://github.com/bsc-pm/nanos6, accessed: 2019-03-24
5. David A. Wheeler: SLOCCount: More about COCOMO, https://dwheeler.com/sloccount/sloccount.html#cocomo, accessed: 2021-07-05

6. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: A proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters **21**(2), 173–193 (2011)
7. Intel: Intel C++ Compiler 19.0 Developer Guide and Reference, https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-cilk-for, accessed: 2019-03-24
8. Luebke, D.: Cuda: Scalable parallel programming for high-performance scientific computing. In: 2008 5th IEEE international symposium on biomedical imaging: from nano to macro. pp. 836–838. IEEE (2008)
9. Nguyen, V., Deeds-rubin, S., Tan, T., Boehm, B.: A sloc counting standard. In: COCOMO II Forum 2007 (2007)
10. Nvidia: CUDA C++ Programming Guide 11.4 (06 2021), https://docs.nvidia.com/cuda/archive, accessed: 2021-07-05
11. OpenMP Architecture Review Board: OpenMP Application Programming Interface 4.5 (11 2015), accessed: 2021-02-18
12. OpenMP Architecture Review Board: OpenMP Application Programming Interface (11 2018), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, accessed: 2019-03-24
13. Perez, J.M., Beltran, V., Labarta, J., Ayguadé, E.: Improving the integration of task nesting and dependencies in OpenMP. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 809–818. IEEE (2017)
14. Sala, K., Bellón, J., Farré, P., Teruel, X., Perez, J.M., Peña, A.J., Holmes, D., Beltran, V., Labarta, J.: Improving the interoperability between MPI and task-based programming models. In: Proceedings of the 25th European MPI Users' Group Meeting. pp. 1–11 (2018)
15. Sala, K., Teruel, X., Perez, J.M., Peña, A.J., Beltran, V., Labarta, J.: Integrating blocking and non-blocking MPI primitives with task-based programming models. Parallel Computing **85**, 153–166 (2019)
16. Shirako, J., Unnikrishnan, P., Chatterjee, S., Li, K., Sarkar, V.: Expressing doacross loop dependences in openmp. In: 9th International Workshop on OpenMP, IWOMP 2013. vol. 8122, pp. 30–44 (09 2013). https://doi.org/10.1007/978-3-642-40698-0
17. Watson, A.H., McCabe, T.J.: Structured testing: A testing methodology using the cyclomatic complexity metric. Tech. rep., NIST Special Publication 500-235 (1996)
18. Wheeler, David A.: SLOCCount, https://dwheeler.com/sloccount, accessed: 2019-03-24
19. Yin, Terry: Lizard, https://github.com/terryyin/lizard, accessed: 2019-03-24