EXPLORING THE BENEFITS OF TASKING IN DACE

Aidyn Aluadin, Ziqiao Kong, Hugo Queinnec, Omkar Zade, Jingyi Zhu

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

DaCe is a data-centric programming environment that parses Python code into an intermediate representation called Stateful Dataflow multiGraph, optimizes it and generates parallel C++ code using the OpenMP API. Originally, DaCe only supports the OpenMP `for` construct. We extended DaCe with the `task` construct[1] and provided examples where code generated using `task` outperforms that using `for`. We also explored the possibility of code generation using task dependencies.

1. INTRODUCTION

Rapid development of multicore systems is increasing the complexity of writing performant code. While Python has become the go-to programming language among domain scientists, it has limitations in taking advantage of parallelism due to its Global Interpreter Lock. Although programmers can enable parallelism in Python with modules such as multiprocessing, it takes extra effort to modify the original code. Many tools have emerged to offer an easier way to parallelize the program. DaCe is one of those frameworks that introduce minimal change to the program. Our project focuses on extending DaCe with the OpenMP `tasking` backend.

Motivation. Currently, DaCe uses the `for` construct of the OpenMP API as its backend. However, there is another useful construct—`task`—which is not utilized in DaCe yet. While `for` is more commonly used in the domains which DaCe aims to support, there are cases where tasking performs better. In the case of heterogeneous workload among iterations of a for loop, tasking might yield a better performance because it dynamically schedules the tasks and thus can better utilize idle threads. Since DaCe uses a graph-based intermediate representation of the program, we can also apply tasking by analyzing the overall structure of the graph and enclosing some subgraphs into tasks which are then executed in parallel. Tasking can also natively support recursive programs such as quick sort [1] or fibonacci numbers [2], which is an advantage over `for`. But, we do not consider such programs because DaCe is not designed to support recursive functions.

Contributions.

- We added a tasking backend that can be used instead of the default `parallel for` backend.
- We introduced tasking on a graph level where subgraphs, in addition to nodes and scopes, can be turned into tasks.
- We evaluated our generated tasking code using NPBench benchmarks.
- We attempted fine-grained tasking with data dependencies but found limitations in the frameworks.

Related work. Legion [3] is a programming model that uses "logical regions" to specify data dependencies between tasks. The tasks are scheduled by their custom parallel scheduling algorithm. The programmer needs to define data regions and access rights which are then used by Legion to execute tasks in correct order. While the idea of applying tasks is similar, our project does not identify or build data dependencies. Dask [4] is a Python library which offers a convenient way to parallelize the code and scale it among multiple clusters. Similar to DaCe, it also uses a graph-based representation to depict dependencies between tasks and overall dataflow. Dask's scheduler and parallel data structures were mainly implemented in Python using the `concurrent.futures` module. In contrast, our project uses OpenMP compiler directives on top of the translated C++ code.

2. BACKGROUND

In this section we briefly go over the concepts, frameworks and tools that we use such as DaCe and OpenMP.

Dataflow. Dataflow refers to the movement of information from one entity to the other in a given environment. For example, information exchange between different clusters

[1] Our code is accessible on GitHub.

in a data center can be considered as a dataflow. More often, dataflow is depicted as a directed acyclic graph (DAG) where nodes are some kind of computation containers. The computation containers take input and produce output which are then passed to other containers.

DaCe. DaCe [5] is a parallel programming framework that can translate code written in high-level language into optimized machine instructions that can be run efficiently in different hardware architectures. It uses a powerful intermediate representation—the Stateful DataFlow multiGraph (SDFG). The SDFG separates the concern between writing scientific code and writing high-performance code tailored to specific hardware. In SDFG, parallel regions of code are represented as Map scopes that have entry and exit nodes and that can contain any subgraphs in between. These subgraphs are self-contained in a way that the external data only comes from the entry/exit nodes of the scope. Thus it is easier to reason about the data dependencies and overall structure of the graph.

OpenMP. OpenMP [6] is an API used for writing parallel programs that are usually executed on a single machine with shared memory architecture. It uses a fork-join model, spawning a team of threads and joining them back throughout the execution of the program. OpenMP provides a set of compiler directives by which a programmer can specify the parts of the code they want to parallelize. A common construct is the `for` construct used to parallelize sequential for loops. It splits the loop iterations into equally sized chunks, assigns them to the team of threads and implicitly waits at the end of the loop until the execution is fully done. There is another OpenMP construct called `task`: a single thread generates tasks according to task regions. Tasks are stored in a queue and executed as per availability of the threads. Using tasks might be beneficial compared to `for` in the case of heterogeneous workload. The `for` construct can produce a schedule where one thread takes much longer to finish its chunk of computation and subsequently halts the others with a lighter workload. This is easily mitigated using tasks as the big workload is distributed more evenly among the threads depending on their availability. However, using tasks is not always better than parallel for loops because there is an additional overhead due to task generation, storage and scheduling. Currently DaCe only uses the `for` construct for executing the parallel regions of the code. We would like to integrate OpenMP tasking to DaCe and attempt to generate C++ code with higher performance.

3. TASKING IMPLEMENTATION

3.1. Straightforward approach to Tasking

Our initial approach to integrating tasking in DaCe was to transform each iteration of a map into a task. Instead of assigning a range of iterations to each thread, tasking allows each thread to choose a task (an iteration), execute it, and return to the task pool for another task. However, we found that this approach was too fine-grained and led to significantly worse performance on almost all benchmarks. Further analysis revealed that the slowdown was mainly caused by the tasking overhead, but also to a lesser extent by false sharing. We obtained better performance only when the iterations are highly heterogeneous (as detailed in section 4.2).

3.2. Advanced Tasking through Manual Transformations

In DaCe, there are instances where multiple independent scopes must be computed in order to generate a final result that depends on them. Each scope can already be parallelized across all available cores, for example in cases where the scope is a map operation. In the current implementation, these scopes are computed one by one, sequentially, until the final result can be calculated. The main idea behind the optimization is to exploit the fact that these scopes are independent. Depending on the structure of the SDFG, some scopes "at the same level" may be able to be computed in any order, or even in parallel. This is where tasking comes into play. It is intended for cases where we have independent tasks that can be executed in any order.

This optimization is not expected to lead to groundbreaking performance improvements as scopes are already independently parallelized. However, there may be some cases where using sequential scopes would be suboptimal, and the use of tasking could lead to a slight performance gain.

Running example. To illustrate how tasking can lead to performance improvements, we will consider a simple SDFG called "multiple maps" (shown in figure 1). It consists of m independent maps at the same level, each with slightly different ranges[2], whose results are all used in an operation at the end. For the purpose of this simple illustration, we will use $m = 4$ maps. In the original implementation of DaCe, map 1 is computed (parallelized across multiple threads with `omp parallel for`), followed by maps 2, 3, and 4 (one map after the other), and finally the final result is computed and outputted. With our custom tasking optimization, we will treat each map as a single threaded

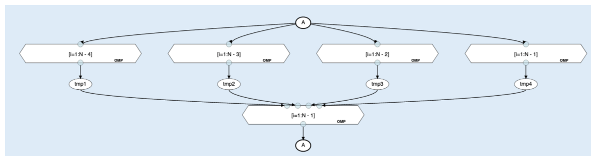

Fig. 1. SDFG of "multiple maps" for $m = 4$

[2]The slightly different ranges are included to make the m maps "not trivially mergeable" into a single map. In practice, if there are several independent maps at the same level in an SDFG, it is indeed because they cannot be merged into a single one.

task. Therefore, if we are parallelizing across 4 threads, each thread will take on one of the 4 tasks, which will all be started at the same time and will execute in parallel.

Performance gains. One theoretical performance improvement in the tasking optimization is what we call "overhead gain". In the original DaCe, as each map is parallelized across all threads, the range of iterations must be divided between threads for each map (shown in figure 2). This overhead time increases linearly with the number of maps: $O(m)$. With the tasking optimization, the first thread creates all tasks corresponding to all maps. No matter the number of maps, this step is done only once and takes $O(1)$ time[3]. Then, each available thread takes a task from the pool of remaining tasks until all tasks are finished[4] (shown in figure 3). Overall, this implementation difference justifies the expectation of some "overhead gain" in the tasking implementation, which will increase with the number of maps. It is possible for the tasking implementation to have some underutilization at the end due to a single thread executing the last map alone. In the worst case, this will last for the duration of one map computation ($O(1)$ in the number of maps). This loss is relatively smaller the more maps there are, and can also disappear when the number of maps is divisible by the number of threads. Additionally, the smaller the map, the relatively larger the overhead for `omp parallel for` becomes, so the tasking implementation should be comparatively faster.

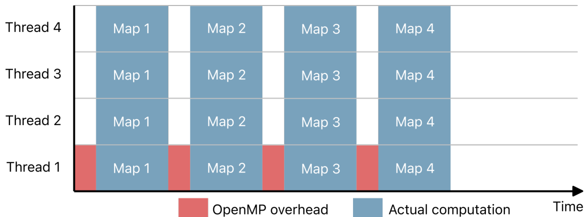

Fig. 2. Simplified execution of "multiple maps" with original DaCe (`omp parallel for` backend), and $m = 4$

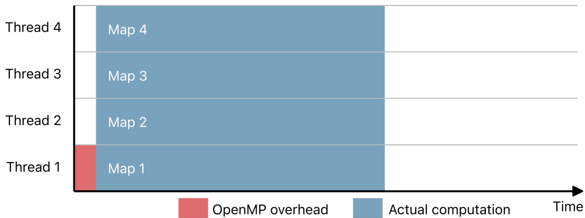

Fig. 3. Simplified execution of "multiple maps" with tasking optimization (each map is a task), and $m = 4$

A second theoretical performance improvement in the tasking optimization is the "barrier gain". In the previous example, we have implicitly assumed that the maps are homogeneous[5]. However, in practice, maps can be heterogeneous, meaning that the computation time depends on the iteration (shown in figure 4). As `omp parallel for` implies an implicit barrier at the end of the map, all threads will have to wait for the longest thread to finish, leading to low utilization. This does not occur with the tasking implementation because each map runs on a single thread. Therefore, we expect some "barrier gain" in the tasking implementation, which will increase with the heterogeneity of the maps. In cases of heterogeneity, it is possible for the `omp parallel for` implementation to actually get faster with an increasing number of threads, even with more threads than CPU cores[6]. Therefore, the barrier gain decreases with the number of threads.

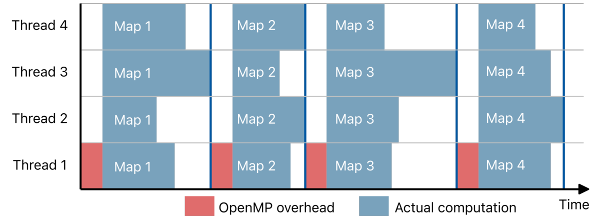

Fig. 4. Simplified execution of an heterogeneous version of "multiple maps" with original DaCe, and $m = 4$

Decision making. Potential "overhead" and "barrier" gains depend on various parameters, such as the number of tasks, number of threads, size of maps, and heterogeneity of maps. It is then up to a performance expert to decide if our optimizations are worth using . Our experimental results (in 4.1) will demonstrate cases where the tasking implementation actually improves performance.

3.3. Code Generation

To transform some functions to high performance computation code, DaCe follows a similar design as LLVM, by splitting the codebase into the frontend, intermediate representation, optimization pass, and code generation backend. This also largely simplifies our work because we only need to deal with the SDFG, the intermediate representation, and then it will work for all frontends. In particular, we focus

[3] It takes slightly longer to create an increasing number of tasks, but this is negligible compared to the original implementation for a number of maps that is much smaller than the number of iterations of each map.

[4] It is intuitively supposed that the overhead time for a thread to take a task is much smaller than the time to divide a range of iterations, and is therefore negligible (this assumption is experimentally verified in 4.1).

[5] Meaning that all iterations have the same computation time, which benefits the `omp parallel for` implementation by leading to almost full utilization of threads.

[6] Because adding more threads divides the work more, so the case where a single thread has longer iterations than all others becomes less likely.

on the *Map* nodes of an SDFG, which represents the actual computation.

Straightforward approach. By default, DaCe generates code using OpenMP `parallel for`, as listing 1 shows.

```
#pragma omp parallel for
for (i = 0;i < n;i ++) {
    // Actual computation
}
```

Listing 1. Parallel for

This allows OpenMP to choose the proper number of threads to split the loop and therefore, a very straightforward approach is to generate 1 task per iteration and synchronize the state until all tasks have stopped, as given in listing 2.

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
        for (i = 0;i < n;i ++) {
            #pragma omp task
            {
                // Actual computation
            }
        }
    }
    #pragma omp taskwait
}
```

Listing 2. Naive approach

Manual transformation. DaCe also offers the ability to transform the SDFG in any way users prefer, because the default code generation backend only guarantees the correctness of the original semantics instead of the best performance. Regarding the OpenMP tasking, we expect the heterogeneous workloads to be more efficient, therefore we also implement a few interfaces to enable users to specify the task boundaries and combine any amount of computation into a single task. Listing 3 gives an example of the generated code from manual SDFG transformation.

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
        #pragma omp task // Task boundary
        {
            for (i = 0;i < n;i ++) {
                // Some computation
            }
            // No sync because no data dependency
            for (j = 0;j < m;j ++ {
                // Some other computation
            }
        } // Task boundary
    }
    #pragma omp taskwait
}
```

Listing 3. Manual transformation

Integration. In order to ensure smooth integration with the existing DaCe codebase, we seamlessly integrated our implementation, rather than relying on a hacky approach. For example, since the naive implementation conflicts with the default `parallel for` backend, we provide a config option *openmp_tasking* to control the behavior. Moreover, we implemented a *Transformation* which matches subgraph structures which can be potentially transformed to tasks, and let the user apply it with a single click via the Visual Studio Code SDFG extension.

3.4. OMP Task Depend

So far, we have been synchronizing tasks at the end of each parallel Map scope to ensure correctness. Data dependencies between tasks is another intuitive way to express parallelism in dataflow graphs. OpenMP version 4.0 and above provide a `depend` clause for the `task` construct to track data dependencies between tasks. In this section, we explore the possibility and limitations of DaCe code generation using OpenMP `task depend`.

Theoretical benefits of data dependency tracking. In the dataflow model, it is theoretically feasible to express parallelism without explicit barriers. In a task-based dataflow model, each task can specify its input and output data. The granularity of parallelism can be adjusted by the size of the tasks. Whenever the data dependencies of a task are fulfilled, the task is ready to be executed. Correct data dependency resolution ensures the correctness of program execution. Since explicit blocking synchronization is not required, threads can be dynamically scheduled in a work-conserving way.

Generate task depend code in DaCe. Each state in an SDFG is a DAG of scopes and tasklets (altogether referred to as code blocks). The in and out edges of each code block represent the input and output data which the code block depends on. In OpenMP, programmers can specify the input and output data dependencies of each task by adding `depend (dependence-type : locators)` after `#pragma omp task`. The locators are a list of non-overlapping memory locations. Here, we focus only on Map scopes, in other words, for loops. In an SDFG, the input and output variable names are usually stored in the edge objects. In a simple case, after the SDFG is fully transformed, we can traverse the SDFG to retrieve the input and output variables of the Map scopes and store them in the MapEntry nodes as in- and out-locators. During code generation, we can directly add `depend (in : in-locators)` and `depend (out : out-locators)` to the tasks.

However, limited support for task-based data dependency in OpenMP and DaCe imposes too many constraints on code generation, making it hard to gain the theoretical benefits in practice.

Limited support in OpenMP. 1) OpenMP runtime only tracks data dependencies between sibling tasks or loop iterations. For example, to specify that the inputs of task C depend on the outputs of two independent tasks A and B, tasks A and B should be nested in a parent task AB. This task AB should be adjacent to task C, and should have in- and out-dependencies of both task A and task B. That restricts task dependencies to following program order. 2) `depend` can not take struct members as locators. Although we can use global arrays or variables instead of structs in DaCe generated code, the lack of support for struct members limits the flexibility of code generation.

Limited support in DaCe SDFG. 1) Variable names are decided during DaCe code generation. After the code of a MapEntry containing `omp task depend` is generated and written, variable names at the corresponding MapExit can still change under multiple conditions. It is currently complicated to modify that part of the DaCe infrastructure. 2) SDFG does not differentiate to-be-nested tasks (e.g., task A and task B described above) from other tasks. It is complicated to track all the to-be-nested tasks during code generation. We could introduce an new graph component during SDFG generation to wrap to-be-nested subgraphs. However, since OpenMP is not the ideal API for the task-based dataflow model, we leave the modification to future work.

4. EXPERIMENTAL RESULTS

4.1. Performance results of Multiple Maps [7]

Experimental setup. We used a 2020 MacBook Pro with an Intel Core i5-1038NG7 at 2 GHz, with 4 cores and hyper-threading enabled, 16 GB memory, running macOS 13.1, Python 3.8.5 and Numpy 1.23.5. Measures were taken using the "profiler" tool included in DaCe.

Parameters. We chose a map of size 1 million, because when the map was too small, the computation was faster on a single core due to the lack of parallelization overhead and cache invalidation. On a 4-core machine, we expect to have the best performance with 4 threads. However, we also explored a range of 1 to 12 threads to see the effect of hyper-threading. We used 16 maps because it is evenly divisible by 4, and 1000 runs for each measure to reduce the standard deviation.

Results[8]. In the homogeneous experiment (figure 5), we found that for most optimizations, the runtime got faster when increasing the number of threads from 1 to 3-4, which shows that our example benefits from parallelization. The execution time was relatively constant above 4 threads, indicating that hyper-threading did not provide any advantage in this case. We notice that our tasking optimization made the execution faster for any number of threads compared to the original DaCe. The same order of improvement was also seen after applying the (powerful) auto-optimizer. Ultimately, our fastest runtime (tasking + auto-optimized on 4 threads) was 6.7% faster than the best result possible before (auto-optimized on 12 threads). This comparison demonstrates the "overhead gain" that we expected.

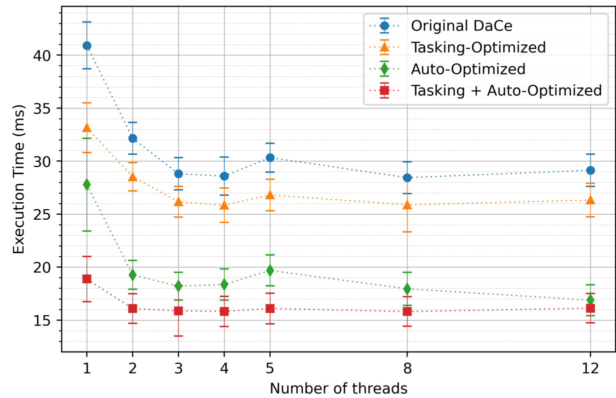

Fig. 5. Execution time of "multiple maps" (homogeneous) for an array size of 1M, depending on the number of threads and optimizations

For the second experiment, we introduced heterogeneity into maps, as described in listing 4.

```
for i in dace.map[1:N - 1]:
    tmp[i] = np.exp(np.sqrt(np.log((A[i - 1] - 1
    * A[i] + A[i + 1])**3))) if i < N/4 else 0
```

Listing 4. Code for a heterogeneous map where the first quarter of iterations takes longer than the following ones

We found (in figure 6) that our tasking optimization was much more powerful and even beat the auto-optimizer from 1 to 5 threads. The original DaCe and auto-optimized version continued to get faster with a higher number of threads, despite the expected hyper-threading overhead. This was because using more threads divided the work more, giving us several threads doing the "slow batch" of iterations instead of just one (which happened with 1 to 4 threads). Ultimately, our fastest runtime (tasking + auto-optimized on 3 threads) was 21.5% faster than the best result possible before (auto-optimized on 12 threads). This comparison demonstrates the "barrier gain" that we expected.

[7] Experiments conducted on "multiple maps" example from section 3.2. The result graphs show averaged measurements over 1000 runs with error bars indicating standard deviation. Dotted lines connect measurement points only for improved readability.

[8] The most relevant metric that showcases our contribution is the comparison of the fastest runtime of the auto-optimized SDFG versus the runtime of the SDFG where we first applied our tasking transformation and then the auto-optimizer.

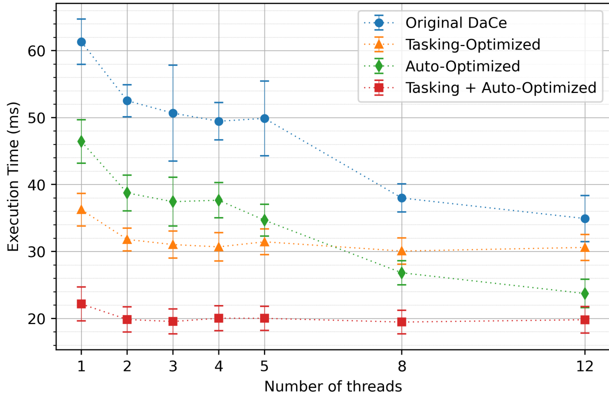

Fig. 6. Execution time of "multiple maps" (heterogeneous) for an array size of 1M, depending on the number of threads and optimizations

4.2. Performance results with NPBench

To get a holistic view of performance of the straightforward tasking implementation, we configured NPBench[7] and ran it with our DaCe tasking implementation. Fig. 7 compares the speedup for various benchmarks on `paper` dataset [9].

Experimental setup. Euler HPC cluster, Intel® Xeon® Gold 5118, 16 cores, 32 GB memory, hyper-threading disabled, Python 3.10, NumPy 1.24, NPBench `@f18e3c7`, Intel MKL 2020 as the BLAS implementation.

Results. We are able to reproduce the speedup of DaCe as described in the paper. For tasking, some of the benchmarks timed out after 200s, so we ask the reader to not interpret the overall speedup (4.2) shown in the heatmap, but observe general trends. We attribute the subpar performance of tasking to two reasons: first, the overhead of task creation and assignment (note that this does not contradict the "overhead gain" in 3.2: as in the straigtforward approach we assign iterations of a map to tasks, while in our novel approach, we assign entire maps to tasks). This also explains the poor scaling of tasking implementation with number of threads. And second, the auto-optimizer in DaCe is designed to work with the `parallel for` implementation, hence the optimizations don't necessarily translate to the tasking implementation.

However, some benchmarks, e.g. `syrk`, perform consistently better with tasking (as described in listing 5).

```
1 def kernel(alpha, beta, C, A):
2     for i in range(A.shape[0]):
3         C[i, :i + 1] *= beta
4         for k in range(A.shape[1]):
5             C[i, :i + 1] += alpha * A[i, k] * A[:
                  i + 1, k]
```

Listing 5. Symmetric rank-k update

This is due to heterogeneous iterations: the amount of work done in the ith outer iteration is proportional to i. As each iteration of outer for-loop is mapped to a task, we get an overall better schedule compared to `parallel for`. Hence, we concluded that naively replacing `parallel for` with tasking isn't promising - which motivated us to implement tasking on the higher i.e. subgraph level.

Benchmarks	parallel for	tasking	numpy
Total	↑11.0	↑4.2	
3mm	↑7.8$^{(2)}$	↑6.8$^{(4)}$	6.96 s
adist	↑19.2$^{(5)}$	↓27.1$^{(4)}$	1.41 s
bicg	↑1.8$^{(17)}$	↑1.8$^{(47)}$	0.28 s
cholesky	↑15.2$^{(1)}$	↑12.5$^{(18)}$	7.22 s
conv2d	↑13.9	↑13.5	18.84 s
doitgen	↑6.3	↑5.3$^{(12)}$	0.52 s
floydwar	↑7.6	↑2.0$^{(14)}$	96.2 s
gemm	↑6.4$^{(10)}$	↑3.5$^{(30)}$	0.53 s
hdiff	↑13.1$^{(2)}$	↑6.4$^{(17)}$	0.4 s
jacobi2d	↑15.7$^{(12)}$	↑3.2$^{(12)}$	136.68 s
lenet	↑1.4$^{(10)}$	↑1.6$^{(30)}$	3.09 s
ludcmp	↑6.0	↑5.0$^{(7)}$	13.56 s
syr2k	↑66.7	↑184$^{(1)}$	18.36 s
syrk	↑82.2$^{(8)}$	↑127$^{(17)}$	6.18 s
trmm	↑72.8$^{(1)}$	↑5.1$^{(6)}$	4.22 s
vadv	↑3.6$^{(9)}$	↓6.7$^{(16)}$	1.41 s

Fig. 7. Speedup of original DaCe with `parallel for` backend vs DaCe with tasking backend vs NumPy (baseline)

5. CONCLUSIONS

In this project, we implemented a tasking backend for DaCe, providing an additional option of generating high performance code with OpenMP `task`. We found patterns in both the code and the SDFG where tasking performs better than `parallel for`. Particularly for heterogeneous workloads, code generated by our tasking backend can potentially lead to a double-digit improvement in performance. A performance expert can easily use our optimizations that are cleanly integrated into DaCe, and can use them from the Python code, or in some cases directly from the Visual Studio Code extension. Additionally, we found that the `task depend` feature in OpenMP is not expressive enough and that code generation using task dependencies would require extensive modification to the DaCe framework.

[9]Only a selected subset is plotted in interest of space.

6. REFERENCES

[1] Ruud van der Pas, Eric Stotzer, Eric Stotzer, and Christian Terboven, *Using OpenMP – The Next Step: Affinity, Accelerators, Tasking, and SIMD*, chapter 3 Tasking, pp. 103–149, 2017.

[2] Sun Microsystems, *OpenMP API User's Guide*, chapter 5.4 Tasking Example, 2009.

[3] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, SC '12, pp. 1–11.

[4] Dask, "Dask documentation," https://docs.dask.org/.

[5] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, SC '19.

[6] OpenMP, "Api specification," https://openmp.org/.

[7] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler, "Npbench: A benchmarking suite for high-performance numpy," in *Proceedings of the ACM International Conference on Supercomputing*, New York, NY, USA, 2021, ICS '21, Association for Computing Machinery.