IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 6 March 2023, accepted 14 April 2023, date of publication 24 April 2023, date of current version 1 June 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3269902

RESEARCH ARTICLE

Optimizing Iterative Data-Flow Scientific Applications Using Directed Cyclic Graphs

DAVID ÁLVAREZ AND VICENÇ BELTRAN
Barcelona Supercomputing Center, 08034 Barcelona, Spain

Corresponding author: David Álvarez (david.alvarez@bsc.es)

This work was supported in part by the European Union's Horizon 2020/EuroHPC Research and Innovation Programme (DEEP-SEA) under Grant 955606; in part by the Spanish State Research Agency—Ministry of Science and Innovation, Generalitat de Catalunya, under Project PCI2021121958 and Project 2021-SGR-01007; in part by the Spanish Ministry of Science and Technology under Contract PID2019-107255GB; and in part by Severo Ochoa under Grant CEX2021-001148-S/MCIN/AEI/10.13039/501100011033.

ABSTRACT Data-flow programming models have become a popular choice for writing parallel applications as an alternative to traditional work-sharing parallelism. They are better suited to write applications with irregular parallelism that can present load imbalance. However, these programming models suffer from overheads related to task creation, scheduling and dependency management, limiting performance and scalability when tasks become too small. At the same time, many HPC applications implement iterative methods or multi-step simulations that create the same directed acyclic graphs of tasks on each iteration. By giving application programmers a way to express that a specific loop is creating the same task pattern on each iteration, we can create a single task directed acyclic graph (DAG) once and transform it into a cyclic graph. This cyclic graph is then reused for successive iterations, minimizing task creation and dependency management overhead. This paper presents the *taskiter*, a new construct we propose for the OmpSs-2 and OpenMP programming models, allowing the use of directed cyclic task graphs (DCTG) to minimize runtime overheads. Moreover, we present a simple *immediate successor* locality-aware heuristic that minimizes task scheduling overhead by bypassing the runtime task scheduler. We evaluate the implementation of the *taskiter* and the *immediate successor* heuristic in 8 iterative benchmarks. Using small task granularities, we obtain a geometric mean speedup of 2.56x over the reference OmpSs-2 implementation, and a 3.77x and 5.2x speedup over the LLVM and GCC OpenMP runtimes, respectively.

INDEX TERMS Taskiter, data-flow programming, ompss-2, openmp, iterative applications.

I. INTRODUCTION

Task-based programming models, pioneered by Cilk [1], have become popular for writing parallel applications since they are better suited than work-sharing models (such as OpenMP's parallel for) to uncover parallelism from dynamic and irregular applications. Generally, these models allow programmers to express parallelism in a tree-like manner, recursively creating tasks. Under this nested-parallel structure, the cost of spawning each task is small, and scheduling can be done optimally through work-stealing.

However, not all parallel applications can be easily written in a tree-like or recursive structure. To give programmers greater flexibility when writing parallel programs, data-flow programming models appeared as a subset of task-based programming. In data-flow programming, parallelism is expressed as a directed acyclic graph (DAG) of tasks, where edges represent dependence relations needed to preserve sequential consistency. Some examples of these programming models include OpenMP Tasks [2], OmpSs-2 [3], PaRSEC [4], StarPU [5], Xkaapi [6] and TBB Graphs [7].

The associate editor coordinating the review of this manuscript and approving it for publication was Claudio Zunino.

Amongst data-flow programming models, some rely on users building the task DAG manually through a special syntax, and then scheduling these DAGs. This paradigm often comes at the cost of having to substantially alter a program to adapt it to the data-flow model.

Other data-flow programming models such as OpenMP and OmpSs-2 tasking rely instead on implicit DAG creation. In this paradigm, users annotate their source code with tasks, and specify their data dependencies. Then, a runtime will build the DAG online, defining dependency relations based on the specified data dependencies. This model provides more productivity than manually defining the task DAG. Furthermore, the runtime can use the provided data dependencies to infer data-locality information, which can then be leveraged during scheduling.

However, the cost of the extra productivity of implicit data-flow programming models is a larger tasking overhead. Creating tasks becomes more complex as data dependencies must be registered and tracked. Moreover, data-flow programs usually have a single thread creating tasks, which can become a bottleneck. Finally, scheduling becomes more challenging as well: as data-flow programming does not necessarily exhibit a recursive task creation pattern, and dependencies can have one-to-many relations, work-stealing scheduling algorithms can suffer from high contention.

In this scenario, data-flow programs must regulate the size of the created tasks to minimize the relative impact of tasking overheads. This forces data-flow programmers to strike a balance in task granularity. We define *task granularity* as the duration of each task in an application [8].

The effects of task granularity on performance have been widely described in literature [9], [10], [11], [12]. Adverse effects are found both when task granularities are too small and when they are too coarse. When the granularity is too small, task creation, scheduling and dependency management become a bottleneck, and tasks cannot be created fast enough to feed all cores. This situation produces two adverse effects that hinder performance: First, some cores remain idle, as not enough work is being created. Second, as the number of tasks ready to execute is very low, there is little chance of applying locality-aware scheduling policies. However, when tasks are too coarse, there may not be enough tasks to feed all cores, the program can suffer from load imbalance, and locality-aware scheduling policies may lose effectiveness as task working sets grow and stop fitting in cache.

Thus, we want to create tasks in a *balanced* region, where granularity is not too fine nor too coarse. This is normally achieved through granularity tuning, but there are relevant situations where tuning is impossible. For example, when the problem size is too small or when scaling out an application. In these cases, it is critical that the runtime efficiently supports small task granularities.

To overcome this issue, data-flow programming models have been optimized over time to minimize these task management overheads [13], [14], [15]. Task creation is generally optimized using scalable memory allocators [16], [17]. Task scheduling is optimized with scalable scheduling techniques, such as work-stealing variants [18] or delegation-based schedulers [13]. Finally, task dependency management requires fine-grained locking or wait-free implementations to achieve good performance. However, these optimizations may not be enough to achieve competitive performance when very fine-grained tasks are needed.

At the same time, many HPC applications present an iterative pattern, creating the same tasks with the same dependencies for each iteration. This results in identical tasks and dependency graphs concatenated one after the other. For example, this happens in iterative methods and solvers, machine learning training phases and multi-step simulations. As such, iterative programs can spend a significant amount of time creating, scheduling and managing tasks and dependencies that are the same for each iteration.

This paper presents and implements two techniques that drastically reduce the main sources of runtime overhead in iterative data-flow applications.

First, we propose a new *taskiter* construct for the OmpSs-2 [3] and OpenMP [2] programming models. The *taskiter* construct annotates loops where each iteration generates the same directed acyclic graph (DAG) of tasks and dependencies. The runtime system then leverages this information to construct a directed cyclic task graph (DCTG) based on the DAG of the first iteration. Dependencies between different iterations are considered and linked in this new directed cyclic graph. In the DCTG, task descriptors and dependency structures are reused for each iteration, drastically reducing task creation and dependency management overheads for any iterations after the first one. The *taskiter* construct does not create any implicit barriers between iterations or after the construct, allowing it to be transparently mixed with successor or predecessor tasks or *taskiter* constructs.

Secondly, we present a new *immediate successor* scheduling technique that preserves data locality while drastically reducing scheduling overheads by bypassing the scheduler. Unlike the *taskiter*, this technique is not restricted to iterative applications.

We note that both proposals can be implemented in other data-flow programming models [4], [5], [6], [19], since the ideas are generally applicable. However, we focus on OpenMP and OmpSs-2 in order to provide a working implementation that can be compared to the current state-of-the-art.

Finally, we will show in the evaluation how both contributions present a particular synergy that results in significant performance improvements for small granularities.

Specifically, our contributions are as follows:

1) We propose the *taskiter* construct for OmpSs-2 and OpenMP to reduce runtime overheads in iterative data-flow applications.
2) We present the *immediate successor* scheduling technique designed to forego most of the scheduling overhead and maximize data locality.

3) We implement both the taskiter construct and the scheduling policy on the state-of-the-art Nanos6 OmpSs-2 implementation, which is competitive with mainstream OpenMP implementations [13].
4) We evaluate the *taskiter* construct on 8 iterative benchmarks and find an average speedup of 3x with a geometric mean of 2.56x for small task granularities.

The rest of this document is structured as follows: Section II reviews the current state of the art, Section III introduces the *taskiter* construct and Section IV introduces the *immediate successor* technique. Then, we evaluate our contributions in Section V, and present the conclusions in Section VI.

II. RELATED WORK

The effects of task granularity on application performance have been thoroughly studied in literature [9], [10], [11], [12]. Moreover, several proposals to reduce task management and scheduling overhead have been proposed.

A. TASK MANAGEMENT OVERHEAD

Tasking overhead can be tackled through granularity tuning, runtime optimizations and model-oriented solutions.

Automatic granularity tuning has been actively researched for task-based programming models. Multiversioning [20] can generate compiler transformations with different task granularities and choose the most appropriate at runtime. Another work explored using cut-off mechanisms [12], where the runtime decides the optimal cut-off point based on a user-provided cost function. Finally, in [21] authors propose an automatic oracle-guided granularity control mechanism for Cilk in which users may elide providing cost functions in some cases.

Some works have focused on optimizations that can be applied to task-based runtimes to reduce synchronization overheads and scale better [13], [15]. Both granularity tuning and runtime optimizations are complementary approaches, which can be combined with model-oriented solutions such as the taskiter.

Other approaches have focused on reducing task overheads by decreasing the total number of tasks that have to be created. Worksharing tasks [22] and Chapel's `coforall` construct [23] can parallelize all iterations from a loop using a single task, reducing their overhead. Similarly, Index Launches [24] can automatically compact several task launches in a loop without need for explicit annotation. Polytasks [25] also merge several similar tasks when they are created at the same time, provided tasks are managed through queues. These approaches reduce the total number of tasks created by an application. In contrast, the proposed taskiter focuses on task reuse, and both approaches can be freely combined, as they are complementary.

In [14], the authors propose the `dep_pattern` clause to cache data dependency patterns reducing dependency management overhead. Our proposal goes further, not only caching dependency structures but preventing task creation altogether. Moreover, the `dep_pattern` clause must be placed on a parent task, which in OpenMP would prevent placing dependencies between iterations to overlap their execution.

Another approach is task DAG caching, provided by the CUDA Graph API [26], which allows GPU programmers to record a graph of kernel invocations and memory copy operations and re-invoke them, removing a significant amount of overhead. The graph API was also motivated by applications with an iterative structure, like machine learning training. However, CUDA Graphs require a barrier between iterations, which prevents the overlap of kernels from multiple iterations and limits the applicability of policies like the immediate successor. Similarly, OpenCL's Command-buffer [27] allows programmers to record a DAG of OpenCL commands and then submit it multiple times in iterative applications. TBB Graphs [28] also allow task graphs that contain cycles, but the programmer must explicitly instantiate all nodes and edges of a task graph manually.

A task DAG caching proposal for OpenMP is the `taskgraph` clause for the `target` and `task` constructs [29]. Similar to CUDA Graphs, authors present a way to record and re-play task DAGs for OpenMP tasks. However, the approach requires task DAGs to be defined inside their own dependency domain with an implicit barrier at the end. This barrier limits the applicability of policies like the immediate successor. Moreover, dependencies between tasks inside the construct and tasks outside it or in other replays are not allowed, breaking the *data-flow* execution model. The `taskgraph` model is a caching strategy and not a task DAG transformation like the one proposed in this paper.

These task caching proposals can potentially improve the performance of iterative applications. However, as we will show in the experimental evaluation, the taskiter construct outperforms task caching approaches.

B. SCHEDULING

Many works have tackled overhead reduction in task scheduling. Scalable schedulers have been traditionally implemented through work-stealing [18], in which each creator thread has a local task queue and can steal from other creators if they run out of work, distributing the scheduling load. Delegation-based techniques [13] are also a suitable implementation, especially on data-flow models where there is often a single creator thread, and thus work-stealing does not offer significant benefits.

Moreover, there have been proposals for locality-aware scheduling, mainly focused on preventing remote accesses on NUMA systems. For example in [30] authors develop a mechanism to accept data distribution hints on `malloc` calls and then leverage data dependency information to determine the best NUMA nodes to schedule a task. These approaches usually improve data locality at the cost of adding some overhead during task submission or scheduling. However, our focus is not on optimal locality but on improving

locality while simultaneously eliminating most of the scheduling overhead.

The philosophy for our scheduling work is similar to Cilk's work-first principle [1]: to remove scheduling overheads from worker threads. However, the heuristic we propose in Section IV is adapted for data-flow applications, works well under any amount of available parallelism, and provides additional locality improvements.

III. THE TASKITER CONSTRUCT

Iterative applications often generate the same dependency graph on each iteration. Dependencies always form a directed acyclic graph of tasks, enforcing restrictions on execution order to maintain serial consistency. Additionally, some task instances from an iteration i will depend on task instances from previous iterations, as we avoid using global barriers. An example of this pattern is shown in the left part of Figure 1, showing two iterations of an iterative application, which models a Gauss-Seidel method with a 4-block matrix. Dependencies between task instances of the same iteration are shown in solid lines, and dashed lines indicate dependencies between iterations.

The taskiter construct is designed to prevent creating and executing the same DAG for each iteration. Instead, the programmer can express that a loop generates the same DAG N times. The programming model runtime will instead generate a directed **cyclic** task graph (DCTG), as shown in the right part of Figure 1. To build the DCTG the runtime executes the first iteration of the loop and generates a regular task DAG. When the first iteration ends, the left and right sides of the DAG are connected, as shown in Figure 1. This representation is then used to execute the remaining $N - 1$ iterations, skipping task creation and significantly minimizing dependency management overheads.

Specifically, dependency management consists of two main components. First, when tasks are created, the runtime must track which tasks are declaring a dependency on each memory location and then apply some logic to transform this information into dependencies between tasks. The second component is dependency release, which tracks the outstanding dependencies for each task, and schedules them when all predecessors have finished. While we cannot remove the overhead of dependency release, as it has to be done for every iteration, we can skip the first dependency management component and calculate the dependencies only in the first iteration.

In essence, we create the task instances and their related data structures once and then reuse the same data structures for all following iterations.

Moreover, the proposed DCTG representation is much more compact in memory than creating the task instances for every iteration. This leads to lower memory usage, which may otherwise be a problem for data-flow programming models when the number of task instances is very large.

This model makes it possible to execute task instances from different iterations simultaneously in a pipelining effect, as the DCTG has no implicit barrier between iterations. This pipelining effect is shown on Figure 2, where thanks to iteration pipelining, the available parallelism is improved. Note that many previous approaches described in Section II did not allow pipelining.

Additionally, dependencies from task instances on the first and last iterations can be matched to tasks outside the taskiter construct, maintaining the data-flow model. Note that the task instances of the first iteration can be executed while building the DCTG, not introducing any performance penalty.

```
1  #pragma omp taskiter
2  for (int timestep = 0; timestep < N; ++timestep) {
3    for (int R = 1; R < numRowBlocks - 1; ++R) {
4      for (int C = 1; C < numColumnBlocks - 1; ++C) {
5        #pragma omp task \
6            depend(in: reps[R-1][C], reps[R+1][C]) \
7            depend(in: reps[R][C-1], reps[R][C+1]) \
8            depend(inout: reps[R][C])
9        computeBlock(rows, cols, R, C, matrix);
10     }
11   }
12 }
```

LISTING 1. Taskiter applied to a sample Gauss-Seidel Heat Equation application. `reps` is a matrix of representatives (one per matrix block).

The syntax of the taskiter construct for OpenMP and OmpSs-2 is defined as the following:

```
#pragma omp taskiter [clause [...]] new-line
    loop
```

```
#pragma oss taskiter [clause [...]] new-line
    loop
```

The *loop* can be any loop statement, provided it fulfills the following conditions:

1) The dependency graph generated by the tasks inside the construct must remain constant for each iteration. However, nested tasks do not have this restriction.
2) The program must remain valid if the code inside the loop body but outside any task is executed only once. This condition can be ignored if the `update` clause is specified, which we explain later on.

The first condition is what the user is actually annotating with the taskiter construct: that the dependency graph for the loop repeats itself and thus can be optimized to a cyclic graph. However, this only needs to be true for first-level tasks (created directly in the loop body), but not for tasks created in deeper nesting levels, allowing irregularity between iterations.

The second condition allows the implementation to execute the loop body only once. Programs can generally be adapted to fulfill this condition by taskifying any code inside the loop body.

For example, we can apply the taskiter construct to an example Gauss-Seidel solver, which iterates through all blocks of a matrix in a wave-front pattern. This results in the code displayed in Listing 1. This code would fulfill the

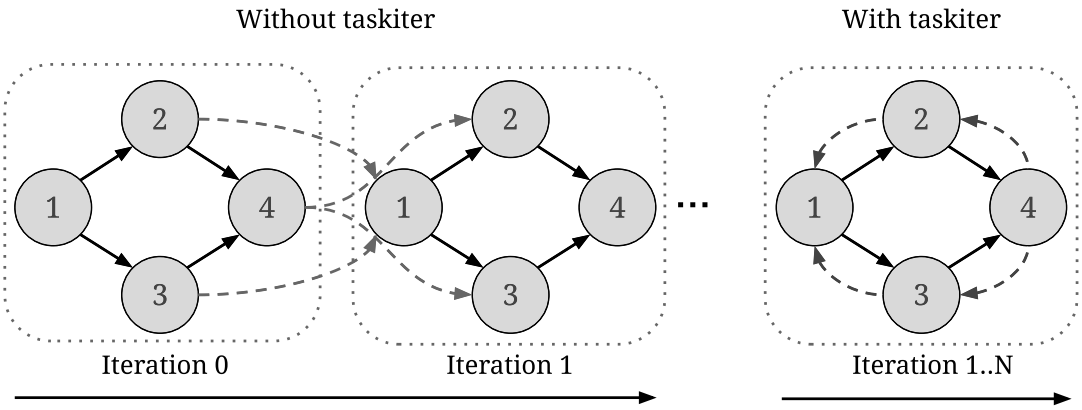


FIGURE 1. Example iterative application, pictured on the left without using the taskiter construct, and on the right when using the taskiter construct.

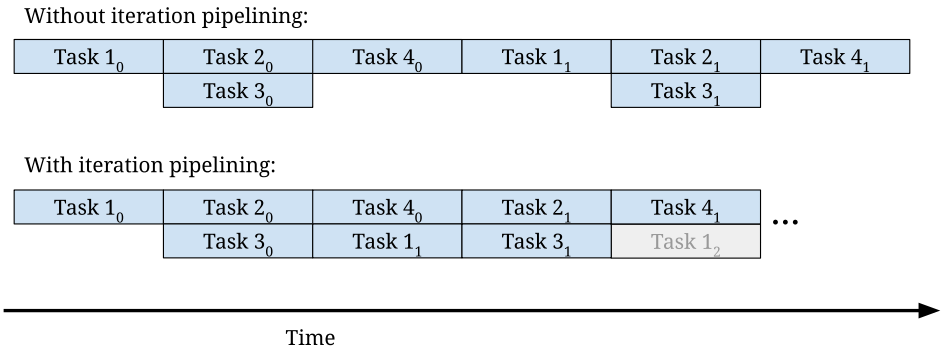


FIGURE 2. Possible execution trace of the application pictured in Figure 1, with and without inserting barriers between iterations. Task i_j refers to task i from the previous figure in iteration j.

requirements of the taskiter, and it only requires the addition of Line 1 from the plain tasks version of this solver.

In the current implementation, users must find suitable loops to apply the construct manually, similar to other OmpSs-2 and OpenMP constructs. The complexity of determining suitable loops to apply the taskiter on varies depending on the specific application. Generally, tracking usages of the induction variable in a loop is a straightforward way to determine if the task DAG changes, and does not require a full analysis of the target application. Moreover, this proposal could be combined with existing static and dynamic analysis tools [31] to facilitate the usage of the taskiter construct.

Two new clauses can be combined with the proposed construct:

- All clauses accepted in the **`task`** construct, since the taskiter is a task on itself.
- The **`unroll(n)`** clause performs loop unrolling, executing the initial `n` iterations instead of one. This clause can be used for loops with a regular dependency graph each `n` iterations. For example, a loop that behaves differently for even and odd iterations can be unrolled two times to generate the cyclic dependency graph. Moreover, with the unroll clause it is possible to have inter-iteration dependencies of distance up to `n`.

```
1  int A;
2  #pragma omp task depend(out: A)
3  A = 1;
4
5  #pragma omp taskiter depend(in: A) depend(out: A)
6  for (int i = 0; i < N; ++i) {
7    #pragma omp task depend(in: A)
8    ...
9    #pragma omp task depend(out: A)
10   ...
11 }
12
13 #pragma omp task depend(in: A)
14 print(A);
```

LISTING 2. Using dependencies between sibling tasks and tasks inside a taskiter region.

- A taskiter with the **`update`** clause will generate a cyclic dependency graph for its tasks only once, but the loop body will be executed for each iteration. Each time the loop body is executed, the parameters used to create each task instance will be recorded, allowing tasks in the generated DCTG to have different parameters for each iteration.

Use of the `taskloop` construct inside a taskiter is allowed, including taskloops with dependencies [32].

The taskiter construct itself can also have dependencies, which can be used to express a dependency from the first

and last iterations of the taskiter to its sibling tasks. This is demonstrated in Listing 2, where using dependencies is convenient because the task in Line 13 can be created before the full DAG of the taskiter in Line 5 is registered.

Task reductions are also supported inside a taskiter region. Nevertheless, OpenMP task reductions imply barriers between iterations, precisely what we try to avoid. To efficiently support reductions inside a taskiter region, in OmpSs-2 we synchronize the combination of reductions with dependencies instead of barriers [33].

The loop transformed by the taskiter does not need to perform a constant number of iterations; thus, executing the next iteration can depend on an arbitrary condition. However, when the loop does not have a run-time constant amount of iterations, the implementation must guarantee that the condition is checked between iterations and the taskiter is stopped when the condition becomes false. Otherwise, when the number of iterations is a run-time constant, the runtime is free to overlap execution of tasks instances from as many different iterations as the dependencies permit.

A. IMPLEMENTATION

When an OmpSs-2 or OpenMP compiler encounters a taskiter construct, it encapsulates one iteration of the following loop as a task. That task is instantiated and passed to the runtime with the number of iterations to execute and a flag indicating it is a taskiter. This special task is queued for execution and will execute the loop's body once, creating any child tasks and registering the initial DAG. However, every child task instance will inherit an iteration counter from the taskiter to track how many times the task instance has to be executed.

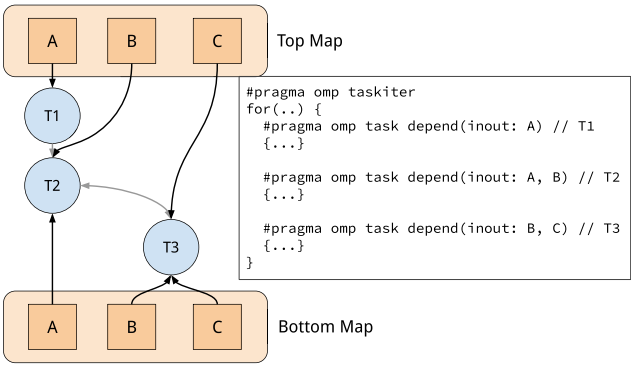


FIGURE 3. Implementation of the taskiter with top and bottom maps. Squares represent entries in the maps, and point to the first/last task to declare a dependency on a specific address. The snapshot of the data structures is taken before the transformation to a cyclic graph.

When the runtime has finished executing the first body of the loop, it will access the *bottom map*, which is a data structure containing the last task that has declared a dependency on each memory location. It will match those tasks to the *top map*, which contains the first task that depends on each memory location. If locations match, there is a dependency from one iteration to the next, and we create an edge between the last and first tasks depending on that location. This edge is marked as crossing the iteration boundary.

An example of the top and bottom map structures is shown in Figure 3. The pictured dependency graph corresponds to the attached code fragment. In this example, for memory location `A`, the top map points to task instance `T1`, which is the first to declare a dependency on `A`. Likewise, `T2` is the last task instance to declare a dependency on `A`. Therefore, there is a cross-iteration dependency where `T1` depends on the previous iteration's `T2`. With these data structures, finding the cross-iteration dependencies is reduced to matching all entries from the bottom map to the ones on the top map.

Whenever a child task finishes, it decreases its iteration counter, and unless it reaches zero, it will try to execute again if its dependencies are satisfied. Each task instance has two data structures that track outstanding dependencies: for even and odd iterations. This way, we do not have to reinitialize the data structures after each iteration. We can track dependencies simultaneously for the current and next iterations without inserting implicit barriers. Moreover, this technique allows us to maintain the wait-freedom of *Nanos6*'s dependency system.

For applications where the transformed loop does not have a run-time constant number of iterations, a special task is inserted in the DCTG, which we call a *control task*. This control task depends on every leaf task, and every root task has a cross-iteration dependency on the previous iteration's control task. Inside the body of this inserted task, we check the loop's condition. If the condition is false, the taskiter is canceled and finishes.

By default, taskiters with control tasks cannot pipeline tasks from different iterations, since the control task serializes iteration execution. However, these control tasks are strided when the taskiter is unrolled, providing means to overlap execution from different iterations. For example, on a taskiter with `unroll(2)`, the control task executed after iteration i does not control the execution of iteration $i+1$, but instead controls the execution of $i+2$. This would allow a rolling window of two iterations being pipelined, and can be increased with larger unroll values.

IV. IMMEDIATE SUCCESSOR

Using the taskiter, we can minimize the overhead of task creation and dependency management. However, reducing those overheads shifts the contention to the remaining source of overhead: task scheduling. After introducing the taskiter in our benchmarks, we observed that the scheduler could become the bottleneck, limiting application performance. Specifically, the speed at which tasks are inserted and requested from the scheduler grows significantly, and so does contention on the locking system of the scheduler. The reference OmpSs-2 implementation currently features a delegation-based centralized scheduler based on [13], but the same contention can be observed in work-stealing implementations when there are few creators.

This section presents a scheduling policy that maximizes data locality for data-flow applications and can be applied without acquiring any scheduler lock. This way, we minimize the number of times any thread has to access the scheduler, reducing contention.

This heuristic is based on a straightforward *successor locality* principle. When one task has a dependency relation with another task, defined by the list of memory locations in their dependency clauses, they probably share a part of their working set. The reasoning behind this principle is straightforward. Data dependencies specify which memory locations a task will access. If two tasks declare a dependency on the same location, both tasks will contain a memory reference to the same location, sharing a part of their working set.

Formally, we define the working set of a task $t_i \in T$ as $W(t_i)$, representing the set of all memory locations that t_i accesses during its execution. Then, we can define a dependency relation, on which a task t_1 depends on a task t_0 as $t_1 \succ t_0$. This denotes constraints in execution order and means that t_1 and t_0 share at least one memory location on the declared data dependencies.

Then, we propose the *successor locality* principle:

$$\forall t_0, t_1 \in T, t_1 \succ t_0 \rightarrow W(t_0) \cap W(t_1) \neq \emptyset$$

Hence, for any pair of tasks t_0 and t_1, if t_1 depends on t_0, the intersection of their working sets is not empty.

While it is possible to create a program on which the above statement is not valid, it matches the patterns observable on most HPC applications written using a data-flow model.

Data locality is paramount when scheduling tasks because it allows applications to exploit the memory hierarchy when the working sets fit in any cache level.

We can leverage this *successor locality* principle to bypass the task scheduler while simultaneously preserving data locality. We do this through the *immediate successor* mechanism, which works as follows:

1) Whenever a task finishes its execution, the worker thread executing it releases its dependencies and can mark one or more *successor* tasks as ready.
2) The first task with the highest priority marked as ready is kept into a local per-worker variable, becoming the *immediate successor*. The priority of a task is calculated from the `priority` clause, which the user specifies.
3) The remaining ready tasks (if any) are placed into the scheduler for other workers to grab.
4) If the worker has an *immediate successor* task, the scheduler is bypassed, and the task is executed next.

This mechanism is illustrated in Figure 4, where a task DAG is introduced, followed by execution traces with and without the immediate successor. The sample task DAG is a subset of a matrix multiplication application. When executing without the immediate successor, each core will execute a ready task, free its dependencies, and then enter the scheduler to find the next ready task in the queue. No priority is given to newly released tasks.

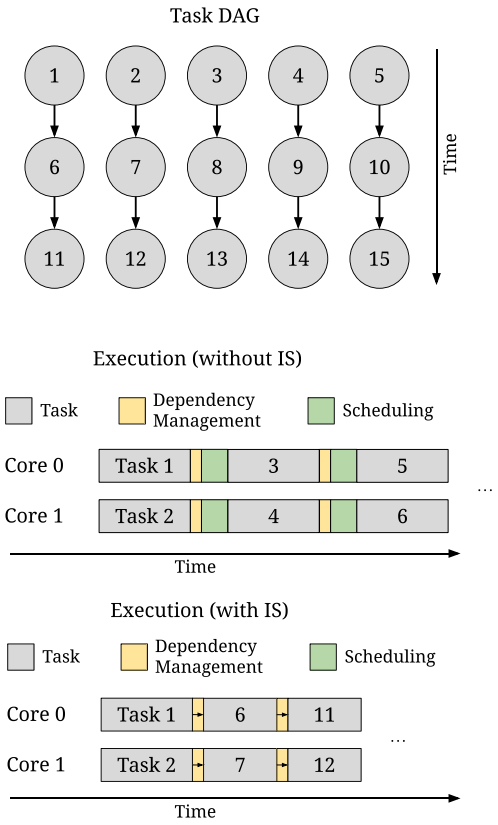


FIGURE 4. Sample task DAG and execution following the immediate successor heuristic.

In contrast, when the immediate successor is enabled, cores will execute the newly ready tasks immediately. For example, when Task 1 ends, Task 6 is marked as ready. As Task 6 is the only task marked as ready, it will be assigned as the immediate successor, and executed immediately without entering the scheduler. Moreover, in the execution shown in Figure 4, the execution time of Task 6 will be reduced due to the presence of part of its working set in cache. Both the removal of scheduling overhead and the increased locality reduce the overall execution time when using the immediate successor mechanism.

Note that we choose the first ready task amongst the ones with the highest priority as the immediate successor. However, choosing the first one is arbitrary, as every ready task follows the successor locality principle.

While this policy is simple, it minimizes the number of times the scheduler is invoked, preventing contention. Moreover, as we show during experimentation, it achieves significant speedups for some applications thanks to its locality-preserving property.

There is a trade-off when applying the immediate successor mechanism. Bypassing the scheduler can be problematic when executing applications that rely on specific scheduling policies (for example, task priorities). We can solve this issue by modifying step 2 of the immediate successor algorithm,

and adding a tunable probability that ready tasks are not marked as immediate successors, allowing threads to enter the scheduler every once in a while.

V. EXPERIMENTAL EVALUATION

To evaluate both the taskiter and the immediate successor (IS) policy, we implemented both features on top of the reference implementation for OmpSs-2. Most changes were located in the *Nanos6* runtime, although we added compiler support for the construct in clang. Our changes in the *Nanos6* runtime only add a small constant overhead to dependency management for tasks inside a taskiter, which does not depend on the number of iterations.

The version of *Nanos6* with support for the taskiter construct is available at `https://github.com/bsc-pm/nodes`, and complementary material such as benchmarks and scripts are located at `https://github.com/dave96/taskiter-ieee-access-2023`.

A. METHODOLOGY

We conducted the evaluation for the taskiter on a node equipped with an AMD EPYC 7742 (Rome) processor with 64 cores clocked at 2.25 GHz and SMT disabled. The system has 1TiB of main memory at 3200MHz. The software stack comprised a CentOS Linux 8.1 distribution with a Linux 4.17 kernel.

We measured the performance when varying task granularity on a set of data-flow benchmarks. We used a combination of smaller benchmarks and larger established HPC applications. The goal is to gather a wide variety of computational patterns representing many scientific applications. Following, we include a brief description of each benchmark and explain their relevance for this experiment:

- The **Multisaxpy** benchmark performs a loop of n Single $A \cdot X + Y$ kernels over two arrays. Each iteration is embarrassingly parallel, stressing mainly task creation and scheduling. Dependent tasks share the totality of their working set, but this working set only fits in cache when task granularity is small. Thus, this application can clearly show the effect of low-overhead locality-aware scheduling policies.
- The acoustic **Full-Waveform Inversion** is a proxy application for exploration geophysics. It implements an iterative method to generate high-resolution subsoil velocity models through collected seismic data. The FWI is divided into a forward propagation and a backward propagation phase, both acting on the modeled three-dimensional soil. Each created task has a large number of many-to-many dependencies, placing stress on dependency management performance.
- The **N-Body** simulation performs several timesteps of the interaction of forces in a particle system. This benchmark is strongly compute-bound, thus data locality is generally not impactful. It places uniform stress on the components of data-flow runtimes, thus providing a reasonable estimate of the amount of overhead introduced.
- The **Heat** Gauss-Seidel equation solver that was showcased in Listing 1. This application is a parallel stencil that displays a wave-front pattern. This implies that the amount of available parallelism varies throughout its execution. As such, this benchmark is sensitive to the introduction of barriers between iterations, as it prevents overlapping the execution of several iterations to hide the parallelism variations. It is strongly memory-bound.
- The **Heat (while)**, is the same application with a variable iteration count instead of a fixed number of iterations. It checks the solution's convergence by computing a residual for each iteration.
- The **HPCCG** is a proxy application for the Conjugate Gradients algorithm, which finds the solution to a sparse system of partial differential equations. It uniformly stresses the components of data-flow runtimes (like the N-Body), is memory-bound, and has 14 different task regions.
- The **HPCG** [34] (High Performance Conjugate Gradients) benchmark, with a fixed iteration count, is an industry-standard benchmark for supercomputers. It features 41 different taskified parallel kernels, together with some wave-front phases. It is designed to reproduce computational and data access patterns representing a wide scientific application set. Moreover, task granularity varies during HPCG's execution, making granularity tuning challenging. This benchmark is sensitive to both data locality and runtime overheads.
- The **HPCG (while)** variant is the HPCG benchmark with a variable iteration count, checking for convergence on each iteration.

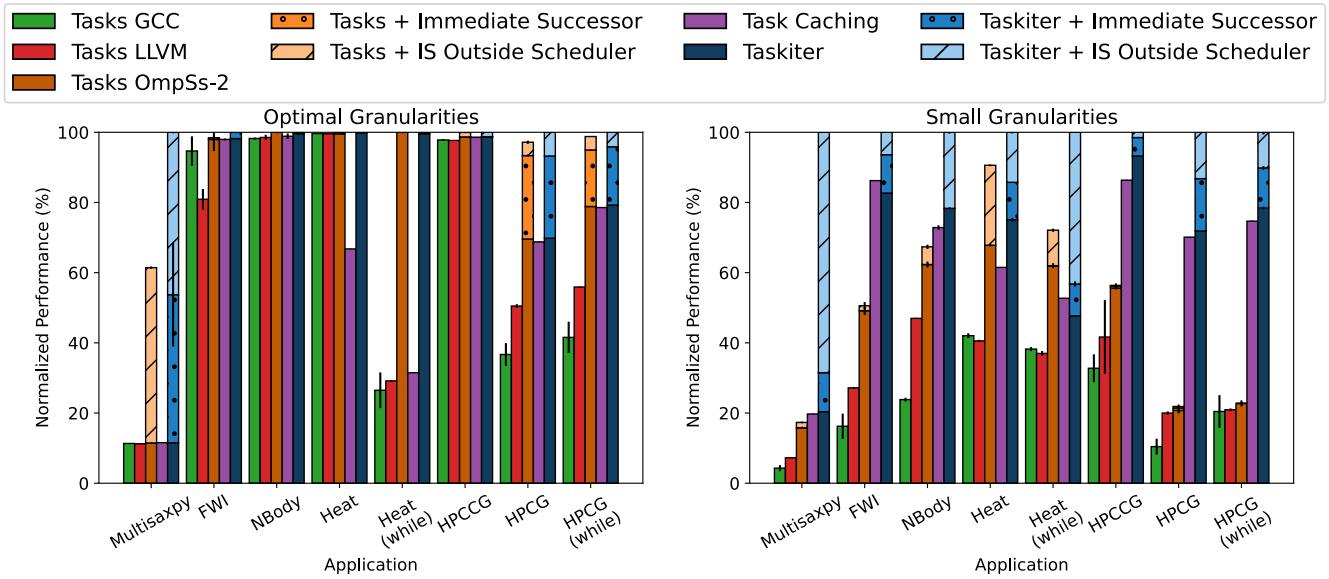
We run two experiments to evaluate the proposed extensions. In the first experiment, we evaluate the performance of our taskiter and immediate successor policy using two task granularities: one where tasks are small, simulating a strong scaling scenario, and another where granularity is optimal. The main goal is to find out if the proposed extensions deliver performance improvements in two scenarios: an optimal case, where granularity tuning has already been manually done for each application, and a case where task granularity is constrained by the problem size or the number of total cores, and thus is inevitably small. We evaluate each proposal in isolation and then combine it with the rest.

In the second experiment, we do a granularity study for each benchmark comparing the optimized *Nanos6* against the reference implementation, other OpenMP runtimes and work-sharing versions of the benchmarks. Both experiments were repeated ten times. In every figure we plot average performance and standard error lines.

Finally, after presenting the results, analyze the HPCG benchmark using execution traces.

B. EXPERIMENT 1: EVALUATION OF PROPOSED EXTENSIONS

In the first experiment, we measure the *normalized performance*, which is the performance of a specific execution



(a) Optimal granularities (b) Small granularities

FIGURE 5. **Performance comparison of different variants when executed with optimal and small granularities.**

relative to the maximum performance of all executions. This normalized performance is obtained based on the figure of merit provided by each application, and absolute performance figures for all granularities are presented later in Section V-C. We run the experiment on two different configurations to observe the most relevant task granularities: First, at the optimal granularity, where performance is in the optimal region. Second, when tasks are too small but still more than 50% peak performance is achieved. This second scenario simulates a strong scaling situation, thus evaluating the scalability of each solution. These granularities were obtained by running a granularity study for each benchmark and selecting: the task size which delivers the maximum performance, and the smallest task size that results in more than 50% of the maximum performance. The complete granularity study is available in Section V-C.

In this experiment, we test seven variants of the *Nanos6* runtime to verify the effect of both the taskiter and the immediate successor policy:

1) *Tasks* is the base OmpSs-2 version of the application, using tasks with dependencies, and no immediate successor.
2) *Tasks + Immediate Successor (IS)* is the same as the *Tasks* version but applies the immediate successor policy. However, this immediate successor is only applied inside the synchronization mechanism for the task scheduler.
3) *Tasks + IS Outside Scheduler* is the *Tasks* version using the immediate successor policy and bypassing the scheduler when possible.
4) *Task Caching* is the application adapted to simulate a task caching approach. We implemented the semantics of the *taskgraph* construct [29], where all data structures are cached between iterations, but without transformation or matching. In other words, it is a *taskiter* with a barrier between iterations. of dependencies between one iteration and the next.
5) *Taskiter* is the application adapted to use a taskiter to transform the main loop into a cyclic graph.
6) *Taskiter + Immediate Successor (IS)* is the same as the *Taskiter* version but applies the immediate successor policy inside the scheduler's synchronization mechanism.
7) *Taskiter + IS Outside Scheduler* is the *Taskiter* version using the immediate successor policy and bypassing the scheduler when possible.

Note that we split the evaluation for the IS policy into two parts. First, we apply the logic behind the immediate successor policy, but every task still has to go through the existing scheduler queues (the *Immediate Successor* version). This way, we can measure when performance increases thanks to better data locality instead of just the reduction of scheduling overhead. In the second part, we also use the immediate successor to bypass the scheduler altogether when an appropriate candidate is found, reducing the contention in the scheduler (the *IS Outside Scheduler* version).

Additionally, we compare every result with two different OpenMP runtimes: the GOMP runtime provided by GCC 10.2.0 and the LLVM OpenMP Runtime on its 13.0.0-rc1 version. We chose to compare against the GCC runtime as a reference implementation for OpenMP, and against the LLVM runtime because it is based on the Intel OpenMP runtime, which is known to have very competitive performance.

Figure 5a shows the performance of each evaluated version versus the maximum figure of merit for each benchmark. Note that we stack the improvements of the immediate successor policies. For instance, the solid orange color bar refers to the *Tasks* version, while lighter orange bars show

TABLE 1. Absolute performance for optimal granularities between the baseline OpenMP and OmpSs-2 versions and the application applying both the Taskiter and the Immediate Successor.

Application	Tasks OmpSs-2		Tasks GCC		Tasks LLVM		Taskiter + IS Outside Scheduler		
Multisaxpy	6.44	± 0.01	6.36	± 0.05	6.31	± 0.01	56.16	± 0.16	Gupdates/s
FWI	22.98	± 0.02	22.21	± 0.98	18.98	± 0.69	23.38	± 0.01	Steps/s
NBody	12664.44	± 19.46	12440.79	± 36.67	12478.56	± 95.54	12656.35	± 43.93	Minteractions/s
Heat	8165.96	± 3.07	8173.86	± 3.70	8169.83	± 1.97	8189.97	± 3.04	Mupdates/s
Heat(while)	8182.78	± 4.17	2167.31	± 415.93	2386.14	± 22.05	8129.20	± 4.06	Mupdates/s
HPCCG	15301.71	± 6.32	15168.81	± 44.43	15147.11	± 12.37	15508.62	± 12.53	MFLOPS
HPCG	16.67	± 0.02	8.79	± 0.78	12.10	± 0.12	23.95	± 0.03	GFLOPS
HPCG(while)	16.16	± 0.02	8.52	± 0.91	11.46	± 0.04	20.50	± 0.03	GFLOPS

how the *Tasks* version performs when combined with the two immediate successor policies. Table 1 summarizes the absolute figures of the OpenMP and OmpSs-2 baselines compared against the best-performing version.

Generally, reducing overheads should have little effect on optimal granularities, but any locality improvements may be noticeable. There are several key insights we can extract from these results. First, if we focus exclusively on the *Tasks* versus the *Taskiter* version, we observe that performance remains very similar. This is expected as these runs happen with optimal granularities, and task creation overhead does not limit performance. However, when we introduce the immediate successor policy, we can achieve higher peak performance on the HPCCG, HPCG and multisaxpy benchmarks, thanks to increased data locality. Moreover, there is a significant difference between placing the IS policy inside and outside the scheduler locking system. This difference is explained by the implementation details of the IS inside the scheduler, where immediate successors are placed in a shared array. When a thread enters the scheduler with no assigned immediate successor, it will try to grab a task from the global queue. If no tasks are found, it will steal another thread's immediate successor, which can have detrimental effects on data locality. When the IS is implemented outside the scheduler, successors can not be stolen by other threads, thus preserving data locality better and resulting in higher performance, even if there was no contention in the scheduler.

For all benchmarks, the performance of the studied versions is either competitive or superior to other OpenMP runtimes.

Figure 5b shows the same results for smaller granularities, where we measure the scalability of each version. Again, we normalized the performance to that of the best-performing version. Table 2 summarizes the absolute figures of the OpenMP and OmpSs-2 baselines compared against the best-performing version. In this case, the *Tasks* version always performs better than other OpenMP runtimes, which confirms that our baseline is already a very scalable runtime. In turn, the *Taskiter* version shows a better performance for small granularities than *Tasks*, thanks to its reduced task creation and dependency management overheads. We find that the most scalable and best-performing version is the *Taskiter + IS Outside Scheduler*.

We can also observe the synergistic effects of both contributions for small granularities. For example, in the HPCG benchmark, the IS policy produces no performance improvement for the *Tasks* version but strongly affects the *Taskiter* version's performance. This is observed in many benchmarks, where the *Tasks + IS Outside Scheduler* has a much smaller speedup than the *Taskiter + IS Outside Scheduler* version due to the synergy between both contributions. Specifically, when not using the taskiter, the immediate successor policy may not be able to find a suitable successor task, since that task may not have been created yet. However, when using the taskiter construct, every task is already created after the first iteration, thus providing more opportunities to apply locality-based policies. We analyze further the HPCG's case in Section V-D.

The *Task Caching* version in optimal granularities either works similarly to the base *Tasks* and *Taskiter* version or causes a slowdown in cases like the Heat equation, where adding a barrier between iterations decreases the available parallelism due to its wave-front pattern. In small granularities, the performance of task caching generally sits between the *Tasks* and *Taskiter* versions as a middle-ground. However, it is outperformed by the construct proposed in this paper in all experiments.

C. EXPERIMENT 2: COMPARISON AGAINST THE BASELINE

The second experiment evaluates performance on a more extensive range of task granularities for the *Taskiter + IS Outside Scheduler* version against both the GOMP and LLVM runtimes and a work-sharing OpenMP version of each application. The work-sharing versions were done using `omp for` constructs on the parallelizable parts of each benchmark, without major code changes except for the Heat application, where the code was adapted to iterate on the matrix's elements diagonally so the loop could be parallelized.

With this comparison, we want to show how the proposed improvements affect the scalability of data-flow applications. Moreover, we will show how the changes can make data-flow programs compete and outperform work-sharing.

Overall, results from Figure 6 show that both the taskiter and the immediate successor policy improve the performance of data-flow iterative applications. We measure task granularity on *instructions* per task, as it is a metric that does not vary between different versions and directly correlates with time. Note that work-sharing versions have a fixed task granularity, and hence are represented as straight lines.

TABLE 2. Absolute performance for small granularities between the baseline OpenMP and OmpSs-2 versions and the application applying both the Taskiter and the Immediate Successor.

Application	Tasks OmpSs-2		Tasks GCC		Tasks LLVM		Taskiter + IS Outside Scheduler		
Multisaxpy	4.63	± 0.07	1.25	± 0.25	2.12	± 0.03	29.25	± 0.06	Gupdates/s
FWI	6.38	± 0.15	2.11	± 0.47	3.53	± 0.03	12.98	± 0.02	Steps/s
NBody	3879.61	± 52.26	1483.39	± 32.62	2923.92	± 11.55	6224.77	± 24.07	Minteractions/s
Heat	3921.10	± 16.25	2430.49	± 40.75	2344.93	± 17.37	5782.37	± 18.00	Mupdates/s
Heat(while)	3737.97	± 42.36	2305.14	± 34.94	2230.07	± 41.02	6030.46	± 7.09	Mupdates/s
HPCCG	7685.61	± 55.13	4527.08	± 548.17	5756.00	± 1453.85	13817.03	± 15.19	MFLOPS
HPCG	3.07	± 0.12	1.54	± 0.33	2.95	± 0.07	14.77	± 0.03	GFLOPS
HPCG(while)	2.79	± 0.11	2.51	± 0.57	2.57	± 0.05	12.26	± 0.08	GFLOPS

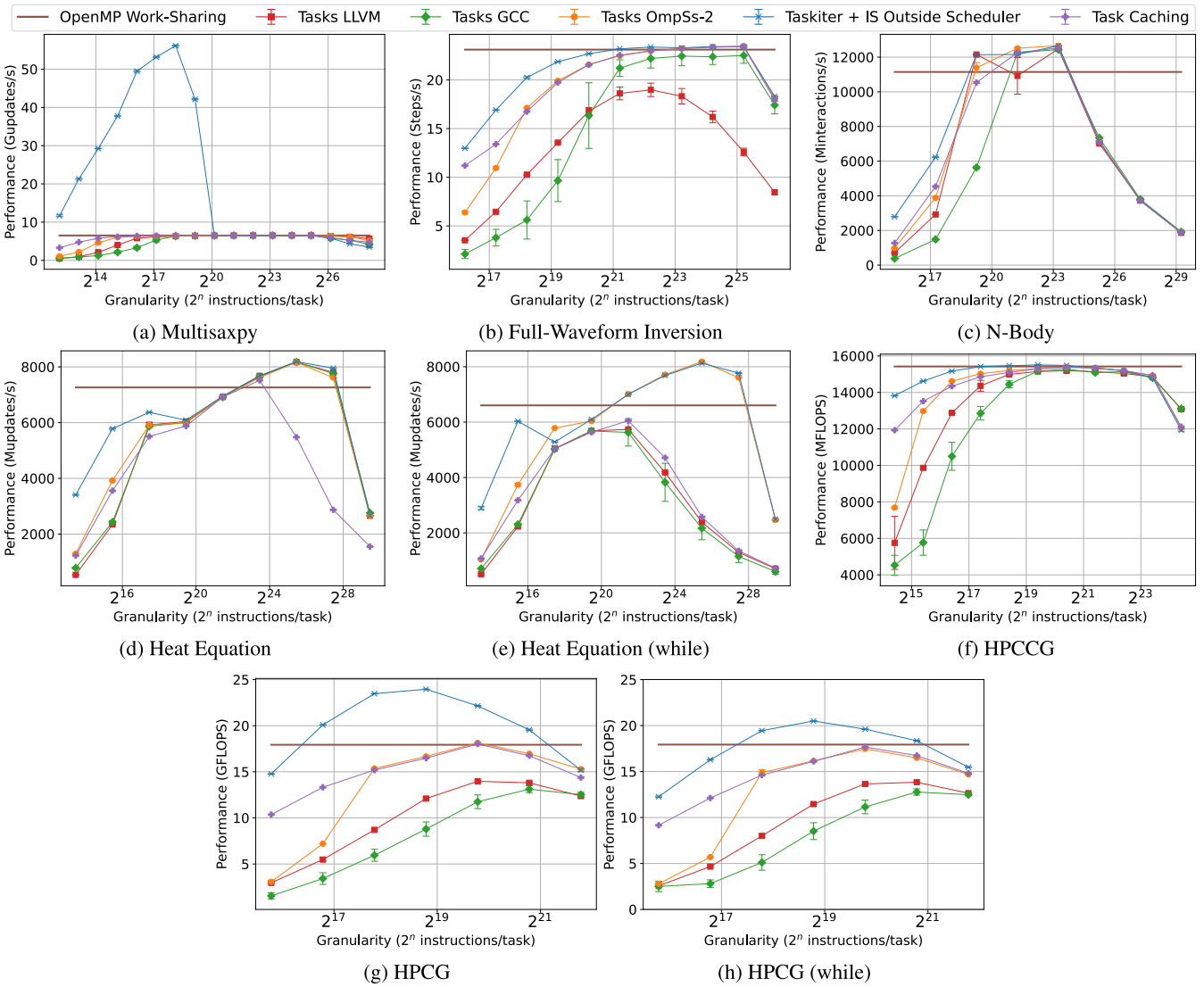


FIGURE 6. Extended experimental evaluation for all granularities on the Taskiter with Immediate Successor against Task Caching, reference OpenMP runtimes and work-sharing.

One result that stands out is Figure 6a, where there is a very notable performance difference for specific granularities in the Multisaxpy benchmark. In this benchmark, starting at a granularity of 2^{19}, the task's working set fits into its L3 cache slice. Therefore, if another task using the same data is immediately scheduled into the same core, all of its working set is hot in cache, which is precisely what the immediate successor policy does. The task creation improvement provided by the taskiter also allows us to apply these locality policies more effectively, producing a synergistic effect. On the other hand, the LLVM and GOMP runtimes do not have this locality policy implemented and schedule other tasks instead, and the work-sharing version has to finish one iteration before the next one starts. We achieve an 8.75x

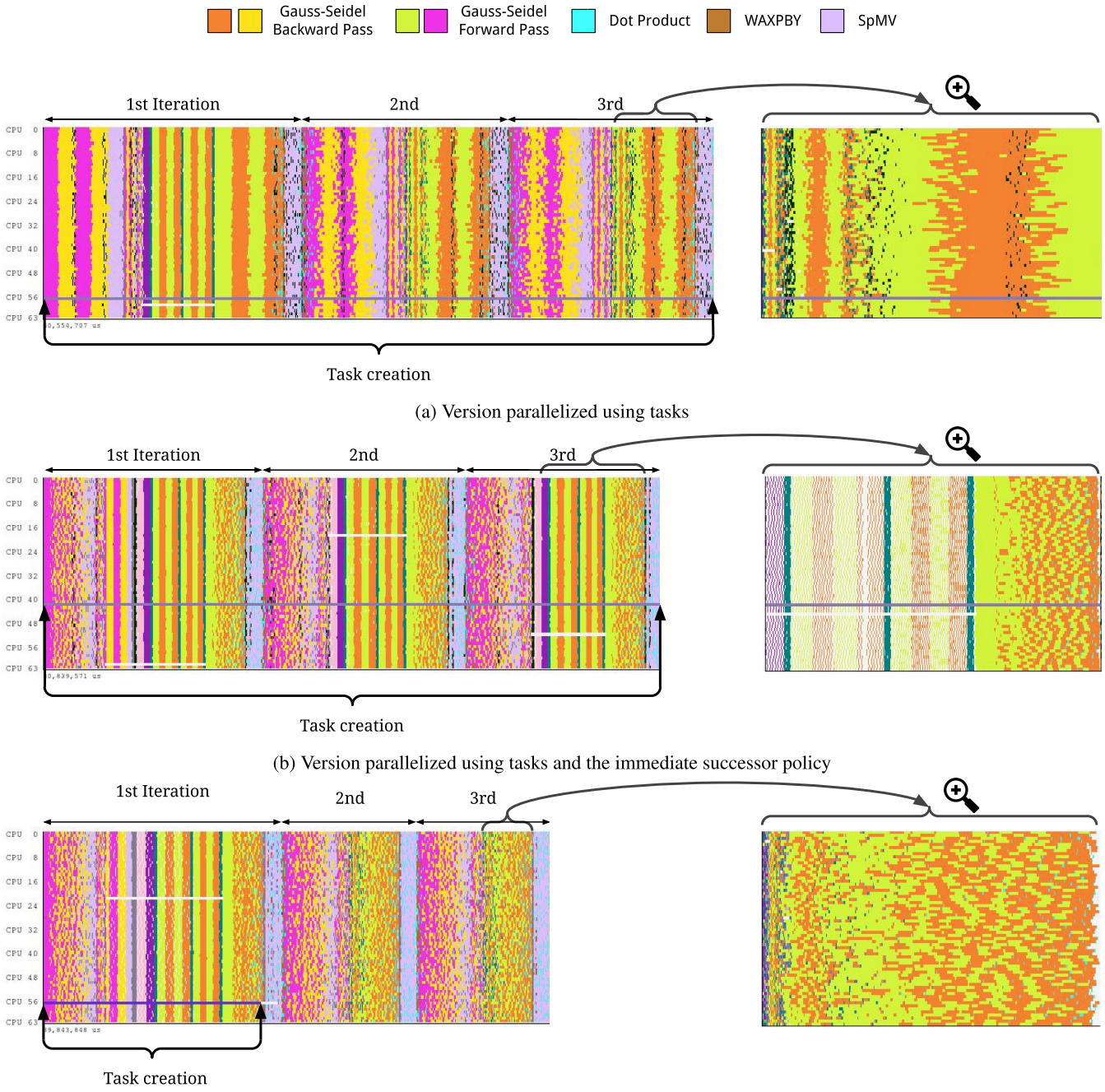


FIGURE 7. Execution traces for the first three iterations of the HPCG benchmark using 64 cores of the AMD machine. The time scale of each trace is the same, a shorter trace implies less elapsed time.

speedup in the optimal granularity compared to the OpenMP baseline.

In the FWI benchmark, shown in Figure 6b, we observe that while the proposed extensions do not significantly improve the performance for optimal granularities, there is a significant improvement when moving onto smaller granularities. LLVM OpenMP shows poor performance executing FWI due to the asymptotical complexity of the dependency registration algorithm, which slows down the creator thread when using many task dependencies. Similarly, for the N-Body benchmark in Figure 6c, the performance improvements are also found in the smaller granularities, and all data-flow implementations outperform the baseline work-sharing version.

Another relevant insight is the Gauss-Seidel heat equation in Figures 6d and 6e. In the first variant, where the number of iterations is fixed, the peak performance obtained is the same for all runtimes, and the difference is seen in the smaller granularities only. However, when we check the residual at each iteration, the performance of LLVM and GOMP drops due to the barriers introduced by task reductions. In OmpSs-2, reductions do not imply barriers, and the implementation of the taskiter allows overlapping the execution of tasks belonging to different iterations, maintaining the available parallelism. This is also why the taskiter is the only version able to outperform work-sharing parallelism in Figure 6e.

The HPCCG proxy application, shown in Figure 6f, shows a similar profile as the Full-Waveform Inversion, as all variants have the same peak performance for optimal granularities, but the proposed extensions allow to maintain that performance when using smaller tasks.

Finally, for the HPCG benchmark, shown in Figures 6g and 6h, only the OmpSs-2 versions are able to match the performance of the work-sharing implementations. Moreover, the *Taskiter + IS Outside Scheduler* is able to outperform all other versions, including a higher peak performance for optimal granularities. OpenMP tasks deliver lower performance than OmpSs-2 in this benchmark due to the barriers that must be introduced when using OpenMP task reductions.

D. HPCG EXECUTION TRACES

So far, we have seen the performance improvements that both the taskiter and the immediate successor policy can deliver. However, we can also leverage the instrumentation included in *Nanos6* to obtain execution traces and study exactly how our contributions affect each application. We chose to study the HPCG benchmark, which is affected by both contributions and showcases its synergistic effects. We obtained execution traces of the application for the granularity highlighted in Figure 6g, and we show these traces in Figure 7.

In all the traces, each row represents one of the 64 cores of the machine. The x-axis represents time, and each color is a different *task type*, which we use to identify different phases of the application. The code color for *task types* is shown in the legend above the traces. Traces should be interpreted in the following manner: in each row, corresponding to a different core, a pixel is colored with the *task type* the core is executing at each point in time. The time scale of each trace is the same, but only three iterations are shown. For each trace, we provide a not to scale zoomed section of a small subset of the execution. We also show with arrows the section corresponding to task creation.

The first trace displayed in Figure 7a shows the baseline tasks version of the HPCG benchmark. Colors denote tasks from different application phases, revealing an iterative pattern. Arrows below the trace highlight the task creator core. This thread executes the *main* task, which creates all other tasks to be executed by the rest of the cores.

When we introduce the immediate successor policy, as seen in Figure 7b, task creation remains the same, but task scheduling changes. In contrast to the well-defined phases on the previous trace, tasks are instead executed in a different order in some instances (see the zoomed-in section). This happens because each orange task depends on a yellow task, and the immediate successor policy decides to schedule one after the other. Note that sections that display this pattern are *shorter* than in the previous trace because better data locality causes tasks to execute faster, as part of the working set is hot in the cache. Moreover, as shown in the zoomed-in section, this produces an unexpected side effect. As tasks execute faster, the task creator cannot keep up and fails to create tasks fast enough to feed all the cores, producing a starvation scenario.

The taskiter solves this starvation problem in Figure 7c. In this case, the task creation is done only during the first iteration, and then a DCTG is constructed, and there is no need to create tasks again. The first iteration is as slow as the other cases, but the following iterations are much shorter because task creation does not limit performance. Moreover, as all tasks are created, we can apply the immediate successor policy more effectively, maximizing locality and exploiting the memory hierarchy better.

VI. CONCLUSION

In this work, we have presented a new directive for OmpSs-2 and OpenMP, the *taskiter*. We have shown how it fits naturally into iterative HPC applications and delivers significant performance gains thanks to reducing task creation and dependency management overheads. We also have combined the *taskiter* with a scalable and straightforward immediate successor heuristic that preserves data locality while reducing scheduling overheads. Moreover, combining both proposals results in a strong synergistic effect, as having faster task creation provides more chances to apply the proposed heuristic.

Our evaluation shows that applying both techniques to data-flow iterative applications delivers significant scalability improvements and speedups, achieving an average speedup of 3x (2.56x geomean) for small granularities compared to the *Nanos6* reference implementation and a 4.62x and 7.02x speedup (3.78x and 5.2x geomean) over the LLVM and GCC OpenMP runtimes, respectively. Moreover, the resulting data-flow applications using the right granularity can compete with or outperform work-sharing on all benchmarks.

The applicability of the proposed approaches is limited to data-flow programming models. Additionally, the *taskiter* is limited to iterative applications with regular or periodic dependency patterns.

In future work, we plan to extend the *taskiter* construct to support device tasks in heterogeneous applications to expand its applicability further.

REFERENCES

[1] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.* New York, NY, USA: Association for Computing Machinery, 1998, pp. 212–223.

[2] OpenMP Architecture Review Board, "OpenMP technical report 8: Version 5.1," Beaverton, OR, USA, Nov. 2010, Accessed: Feb. 1, 2020.
[3] A. Duran, R. Ferrer, E. Ayguadé, R. M. Badia, and J. Labarta, "A proposal to extend the OpenMP tasking model with dependent tasks," *Int. J. Parallel Program.*, vol. 37, no. 3, pp. 292–305, Jun. 2009.
[4] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Comput. Sci. Eng.*, vol. 15, no. 6, pp. 36–45, Nov. 2013.
[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," in *Euro-Par 2009 Parallel Processing*, H. Sips, D. Epema, and H.-X. Lin, Eds. Berlin, Germany: Springer, 2009, pp. 863–874.
[6] T. Gautier, J. V. F. Lima, N. Maillard, and B. Raffin, "XKaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, May 2013, pp. 1299–1308.
[7] J. Reinders, *Intel Threading Building Blocks—Outfitting C++ for Multi-Core Processor Parallelism*. Sebastopol, CA, USA: O'Reilly Media, Jan. 2007.
[8] C. P. Kruskal and C. H. Smith, "On the notion of granularity," *J. Supercomput.*, vol. 1, no. 4, pp. 395–408, Aug. 1988.
[9] A. Rosà, E. Rosales, and W. Binder, "Analysis and optimization of task granularity on the Java virtual machine," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 3, pp. 1–47, Jul. 2019.
[10] T. Gautier, C. Perez, and J. Richard, "On the impact of OpenMP task granularity," in *Evolving OpenMP for Evolving Architectures*, B. R. de Supinski, P. Valero-Lara, X. Martorell, S. M. Bellido, and J. Labarta, Eds. Cham, Switzerland: Springer, 2018, pp. 205–221.
[11] D. Akhmetova, G. Kestor, R. Gioiosa, S. Markidis, and E. Laure, "On the application task granularity and the interplay with the scheduling overhead in many-core shared memory systems," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2015, pp. 428–437.
[12] A. Navarro, S. Mateo, J. M. Perez, V. Beltran, and E. Ayguadé, "Adaptive and architecture-independent task granularity for recursive applications," in *Scaling OpenMP for Exascale Performance and Portability*, B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, and M. S. Müller, Eds. Cham, Switzerland: Springer, 2017, pp. 169–182.
[13] D. Álvarez, K. Sala, M. Maroñas, A. Roca, and V. Beltran, "Advanced synchronization techniques for task-based runtime systems," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.* New York, NY, USA: Association for Computing Machinery, Feb. 2021, pp. 334–347.
[14] A. Podobas, M. Brorsson, and V. Vlassov, "Turbobłysk: Scheduling for improved data-driven task performance with fast dependency resolution," in *Using Improving OpenMP for Devices, Tasks, More*, L. DeRose, B. R. de Supinski, S. L. Olivier, B. M. Chapman, and M. S. Müller, Eds. Cham, Switzerland: Springer, 2014, pp. 45–57.
[15] G. Contreras and M. Martonosi, "Characterizing and improving the performance of Intel threading building blocks," in *Proc. IEEE Int. Symp. Workload Characterization*, Oct. 2008, pp. 57–66.
[16] J. Evans, "A scalable concurrent malloc(3) implementation for FreeBSD," in *Proc. BSDCan*, Apr. 2006, pp. 1–14.
[17] D. E. Berger, S. K. McKinley, D. R. Blumofe, and R. P. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," in *Proc. 9th Int. Conf. Architectural Support Program. Lang. Operating Syst.* New York, NY, USA: Association for Computing Machinery, 2000, pp. 117–128.
[18] J. Yang and Q. He, "Scheduling parallel computations by work stealing: A survey," *Int. J. Parallel Program.*, vol. 46, no. 2, pp. 173–197, Apr. 2018.
[19] F. Lordan, E. Tejedor, J. Ejarque, R. Rafanell, J. Álvarez, F. Marozzo, D. Lezzi, R. Sirvent, D. Talia, and R. M. Badia, "ServiceSs: An interoperable programming framework for the cloud," *J. Grid Comput.*, vol. 12, no. 1, pp. 67–91, Mar. 2014.
[20] P. Thoman, H. Jordan, and T. Fahringer, "Adaptive granularity control in task parallel programs using multiversioning," in *Euro-Par 2013 Parallel Processing*, F. Wolf, B. Mohr, and D. A. Mey, Eds. Berlin, Germany: Springer, 2013, pp. 164–177.
[21] U. A. Acar, V. Aksenov, A. Charguéraud, and M. Rainey, "Provably and practically efficient granularity control," in *Proc. 24th Symp. Princ. Pract. Parallel Program.* New York, NY, USA: Association for Computing Machinery, Feb. 2019, pp. 214–228.
[22] M. Maroñas, K. Sala, S. Mateo, E. Ayguadé, and V. Beltran, "Worksharing tasks: An efficient way to exploit irregular and fine-grained loop parallelism," in *Proc. IEEE 26th Int. Conf. High Perform. Comput., Data, Analytics (HiPC)*, Dec. 2019, pp. 383–394.
[23] B. L. Chamberlain, "Chapel," in *Programming Models for Parallel Computing*, P. Balaji, Ed. Cambridge, MA, USA: MIT Press, 2015, ch. 6, pp. 129–159.
[24] R. Soi, M. Bauer, S. Treichler, M. Papadakis, W. Lee, P. McCormick, A. Aiken, and E. Slaughter, "Index launches: Scalable, flexible representation of parallel task groups," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.* New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 1–14.
[25] D. Orozco, E. Garcia, R. Pavel, R. Khan, and R. Guang Gao, "Polytasks: A compressed task representation for HPC runtimes," in *Languages and Compilers for Parallel Computing*, S. Rajopadhye and M. M. Strout, Eds. Berlin, Germany: Springer, 2013, pp. 268–282.
[26] *Cuda C Programming Guide*, NVIDIA, Santa Clara, CA, USA, 2021.
[27] E. Crawford and J. Frankland, "OpenCL command-buffer extension: Design and implementation," in *Proc. Int. Workshop OpenCL.* New York, NY, USA: Association for Computing Machinery, May 2022.
[28] *OneAPI Threading Building Blocks (OnetBB)*, Intel Corporation, Santa Clara, CA, USA, 2022.
[29] C. Yu, S. Royuela, and E. Quiñones, "Enhancing OpenMP tasking model: Performance and portability," in*OpenMP: Enabling Massive Node-Level Parallelism*, S. McIntosh-Smith, B. R. de Supinski, and J. Klinkenberg, Eds. Cham, Switzerland: Springer, 2021, pp. 35–49.
[30] A. Muddukrishna, A. Peter Jonsson, and M. Brorsson, "Locality-aware task scheduling and data distribution for OpenMP programs on NUMA systems and manycore processors," *Sci. Program.*, vol. 2015, Jan. 2016, Art. no. 981759.
[31] S. Economo, S. Royuela, E. Ayguadé, and V. Beltran, "A toolchain to verify the parallelization of OmpSs-2 applications," in *Euro-Par 2020: Parallel Processing*, M. Malawski and K. Rzadca, Eds. Cham, Switzerland: Springer, 2020, pp. 18–33.
[32] M. Maroñas, X. Teruel, and V. Beltran, "Openmp taskloop dependences," in *Proc. 17th Int. Workshop OpenMP (IWOMP)*, in Lecture Notes in Computer Science, N. Maruyama, B. R. de Supinski, and M. Wahib, Eds. Bristol, U.K., Sep. 2021, pp. 1–15.
[33] F. P. Roca, "Extending ompss programming model with task reductions: A compiler and runtime approach," Bachelor's thesis, Barcelona School of Informatics, Universitat Politècnica de Catalunya, Barcelona, Spain, Jan. 2017.
[34] M. A. Heroux and J. Dongarra, "Toward a new metric for ranking high performance computing systems," Sandia Nat. Lab., Alburquerque, NM, USA, Tech. Rep., Jun. 2013.

DAVID ÁLVAREZ received the bachelor's degree in informatics engineering and the M.Sc. degree in innovation and research in informatics from Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 2019 and 2021, respectively, where he is currently pursuing the Ph.D. degree with the Computer Architecture Department (DAC).

Since 2019, he has been with the System Tools and Advanced Runtimes (STAR) Research Group, Barcelona Supercomputing Center (BSC), focusing on task-based programming models for HPC, low-level parallel runtimes, and heterogeneous computing. He is currently a part-time Lecturer with DAC, UPC.

VICENÇ BELTRAN received the B.Sc. and M.Sc. degrees in computer engineering and the Ph.D. degree in computer architecture from Universitat Politècnica de Catalunya (UPC), Barcelona, Spain, in 2004 and 2009, respectively.

Since 2009, he has been a Senior Researcher with the Barcelona Supercomputing Center (BSC), where he works on parallel and distributed programming models and system software for HPC systems. He has participated in several EU and industrial projects, such as DEEP/-ER/EST and INTERTWinE. He currently leads the System Tools and Advanced Runtimes (STAR) Group that focuses on research crossing multiple layers of the system software stack, from OS, runtimes, low-level APIs to programming models, tools, and applications.