Suspending OpenMP Tasks on Asynchronous Events: Extending the Taskwait Construct

Romain Pereira, Maël Martin, Adrien Roussel, Thierry Gautier, Patrick Carribault

Suspending OpenMP Tasks on Asynchronous Events: Extending the Taskwait Construct

Romain Pereira[1,3], Maël Martin[1,2], Adrien Roussel[1,2], Patrick Carribault[1,2], and Thierry Gautier[3]

[1] CEA, DAM, DIF, F-91297 Arpajon, France
{romain.pereira@cea.fr, mael.martin@cea.fr, manuel.ferat@cea.fr, adrien.roussel, patrick.carribault}@cea.fr
[2] Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation, 91680 Bruyères-le-Châtel, France
[3] Project Team AVALON INRIA, LIP, ENS-Lyon, Lyon, France
thierry.gautier@inrialpes.fr

Abstract. Many-core and heterogeneous architectures now require programmers to compose multiple asynchronous programming model to fully exploit hardware capabilities. As a shared-memory parallel programming model, OpenMP has the responsibility of orchestrating the suspension and progression of asynchronous operations occurring on a compute node, such as MPI communications or CUDA/HIP streams. Yet, specifications only come with the `task detach(event)` API to suspend tasks until an asynchronous operation is completed, which presents a few drawbacks. In this paper, we introduce the design and implementation of an extension on the `taskwait` construct to suspend a task until an asynchronous event completion. It aims to reduce runtime costs induced by the current solution, and to provide a standard API to automate portable task suspension solutions. The results show twice less overheads compared to the existing `task detach` clause.

Keywords: OpenMP · MPI · Asynchronous Programming · Dependent Task

1 Introduction

To increase computational power evermore, supercomputer nodes evolved towards many-cores and heterogeneous architectures. Currently, 2 out of the 3 most powerful supercomputers nodes are made of 64-core processors, 4 GPUs and NICs [4]. In the future, nodes may even become more complex with other accelerators such as FPGAs for their energy efficiency [25]. Programming portable and efficient scientific simulation on such architecture is challenging. Vendors, research laboratory and programmers built *programming models* such as OpenMP, MPI, CUDA, leading users to *compose* multiple programming models to fully exploit compute resources.

[4] https://www.top500.org/lists/top500

Recent work on interoperability with MPI [16, 18, 20, 21] and multi-GPU offload through target tasks [3, 6, 23] shows that OpenMP dependent task model is promising to overlap computation using multiple heterogeneous and asynchronous programming model. However, porting existing applications to an asynchronous dependent task-based model creates more difficulties than historical MPI+OpenMP `parallel-for` programming.

The objective of this paper is to propose standard extensions to suspend tasks until the completion of an asynchronous event. Our contributions are

- (1) A proposal on extending the `taskwait` construct with the `detach(event)` clause, with a standard API, to wait until the completion of an external event,
- (2) Proof-of-concept implementations into LLVM and MPC-OMP available online [5],
- (3) Evaluations showing 2x less runtime overheads over existing solutions in case of low-concurrency (a synchronization followed by a continuation task).

The extension is further-motivated in Section 2. A definition is proposed in Section 3 and evaluations are conducted in Section 4. Related works are reviewed in Section 5 and we conclude this work in Section 6.

2 Motivations

OpenMP tasking model is a portable solution for composing multiple programming models and their asynchronous operations, such as MPI requests, CUDA streams, FPGAs offloads or even disk I/O. Yet, specifications mostly come with the `task detach(event)` clause and the `interop` construct for standard interoperability. We propose to extend the `taskwait` construct with the `detach` clause as an alternative interoperability building block to suspend a task until an asynchronous event completion.

Porting HPC Applications using OpenMP Tasks Early porting of applications to task-based OpenMP showed difficulties in handling asynchronous operations synchronization within tasks. Using the `taskyield` construct in [14], programmers assumed that the thread necessarily switches to another task if one is ready: this is not the case, the standard only specifies it at a scheduling-point, and the implementation decides whether to actually switch tasks. In two other examples, the porting of a Cholesky factorization [22] and a plasma simulation [19] had to sequentialize MPI communications, potentially at the expense of performance, as the standard was not providing any guarantees on suspending tasks when synchronizing on MPI requests.

[5] https://gitlab.inria.fr/ropereir/iwomp23

A Lack of Interoperability These issues lead the community to develop interoperability interfaces between OpenMP and MPI. The Task-Aware MPI (TAMPI) [20] library was proposed as an extra layer above MPI, to overlap MPI request synchronization with useful computation through task-switches. This library must be used by programmers adapting their code to replace MPI with TAMPI calls. MPICH+BOLT [5, 12] through Argobots, or MPC [16], proposed to automate this interoperability through the threading library, so that OpenMP tasks can perform blocking MPI calls and suspend seamlessly. While all these approaches enable working code, OpenMP and MPI standards provided no guarantees on it requiring programmers to manage an additional interoperability layer, and ensure its correct implementation from both runtimes.

Providing Guarantees OpenMP specifications adopted the `detach` clause which provides guarantees to programmers, and enables the overlap of asynchronous operation progression with useful computation through task scheduling, as discussed in [1]. Though, the `detach` clause also impacted other asynchronous programming models. For instance, proposals were made to the MPI specifications to register a callback on request completion [18,21] which are being standardized [6] [7]. This would be used in codes as depicted in Listing 1.1 which we retrieved from [18, 20] as follows:

```
1  omp_event_handle_t ev_handle;
2  # pragma omp task detach(ev_handle) depend(out: data)
3  {
4      MPI_Request req;
5      MPI_Irecv(data, ..., &req);
6      MPIX_Detach(&req, omp_fulfill_event, ev_handle);
7  }
8  # pragma omp task depend(in: data)
9  {
10     work_with(data);
11 }
```

Listing 1.1. task detach(event) approach retrieved from [18, 20, 21]

With the `task detach(event)` approach, application programmers express two dependent tasks for managing asynchronous operations: the launch (line 2) and its continuation after completion (line 8). `MPIX_Detach` (line 6) registers the `omp_fulfill_event(ev_handle)` callback on the MPI request completion, which raises an *allow-completion* event to the OpenMP runtime. This approach does provide guarantees for asynchronous operation overlapping expected by programmers, as opposed to the test-and-yield approach.

Limits of 'task detach' Nevertheless, we argue the `task detach(event)` clause has three drawbacks on the minimal example presented: (a) it implies unneces-

[6] https://github.com/mpiwg-hybrid/hybrid-issues/issues/6
[7] https://github.com/devreal/ompi/tree/mpi-continue-master

sary costs on programming and execution, (b) it is error-prone and (c) it discards the interoperability responsibility to the programmer.

Regarding (a), creating two dependent tasks does not provide more parallelism in this case. Our evaluation section 4.1 shows that our proposal can remove some task management costs by following the C sequential order of execution, and synchronizing at some point in the execution without spawning a new task.

On (b), OpenMP memory model does not mention anything specific on tasks with a `detach` clause. When a C variable is declared as such, we believe the memory model falls back to "*[...] the programmer must use synchronization that ensures that the lifetime of the variable does not end before completion of the explicit task region sharing it. Any other access by one task to the private variables of another task results in unspecified behavior*". In practice, the code Listing 1.1 retrieved from [18, 20] while executes as follows: the variable `req` is pushed onto the executing thread stack (line 4), the asynchronous operation starts (line 5), a completion callback is registered into the MPI runtime (line 6) and the task returns (line 7). The thread may then schedule other tasks until the callback is raised, potentially erasing the `req` variable onto its stack that has been passed by address earlier. The `MPI_Detach` proposal [18] tackles this risk by dereferencing and copying the `MPI_Request` pointer [8]. Note that `MPI_Request` copy is not standardized, even though Open MPI, MPICH or MPC-MPI represents `MPI_Request` as integers so copies are achieved seamlessly. The `MPI_Continue` proposal [21] tackles this risk by consuming the `MPI_Request` setting it to `MPI_REQUEST_NULL` on return.

Finally, for (c), the programmer has to create a continuation task explicitly and remove the synchronization call (`MPI_Wait`). On the other hand, runtime interoperability requiring no user code modification is possible and has been proposed in [12, 16]: as illustrated on Listing 1.2, the MPI runtime will automatically suspend the current OpenMP task so the OpenMP task scheduler can overlap the synchronization line 4 with other tasks.

```
1 # pragma omp task untied
2 {
3     MPI_Recv(&req, ...);
4     work_with(data);
5 }
```

Listing 1.2. MPI and OpenMP runtime interoperability approach

3 Extending Taskwait to Asynchronous Events

We propose to extend the `taskwait` construct with the existing `detach(event)` clause and to provide an `omp_taskwait_detach(event)` routine. It is depicted on Listing 1.3 and fixes drawbacks (a) and (b): only one task and no dependency are needed, and variables pushed to the thread stack may not be erased. The

[8] `https://github.com/RWTH-HPC/mpi-detach/blob/master/detach.cpp#L66`

routine is also a first step towards fixing the issue (c) and *standardizing runtime interoperability*, which is further discussed in Section 4.2. We introduce new elements to the standard specifications.

```
1 # pragma omp task
2 {
3     MPI_Request req;
4     MPI_Irecv(data, ..., &req);
5     omp_event_handle_t ev_handle = omp_task_continuation_event();
6     MPIX_Detach(&req, omp_fulfill_event, ev_handle);
7     # pragma omp taskwait detach(ev_handle)
8     work_with(data);
9 }
```

Listing 1.3. taskwait detach(event) proposal

3.1 Definitions

taskwait construct The `taskwait` construct is extended as follows:

- **Semantics** - if a `detach` clause is present on a `taskwait` construct, the current task region is suspended until the current task continuation event is fulfilled.
- **Restrictions** - The `detach` clause may only appear on a `taskwait` if no `depend` or `nowait` clauses are present.

Combined-clauses restrictions were motivated by the lack of practical examples. If practical use-cases were to be found, we propose to follow the current specifications that replicates empty tasks behaviors:

If the **detach** clause is present on the **taskwait** construct, one or more depend clauses are present and the **nowait** clause is not present, the behavior is as if these clauses were applied to a **task** construct with an empty associated structured block that generates a *mergeable* and *included task*. Thus, the current task region is suspended until the *predecessor tasks* of this task complete execution and its `allow-continuation` event is fulfilled.

If the **detach** clause is present, one or more **depend** clauses are present, and the **nowait** clause is present on the **taskwait** construct, the behavior is as if these clauses were applied to a **task** construct with an empty associated structured block that generates a task for which execution may be deferred, converting the *allow-continuation* event to an *allow-completion* event. Thus, all *predecessor tasks* of this task must complete execution, and its `allow-completion` event must be fulfilled, before any subsequently generated task that depends on this task starts its execution.

omp_taskwait_detach(event) The `omp_taskwait_detach(event)` routine behave as if a `# pragma omp taskwait detach(event)` had been specified.

task continuation (Tasking Terminology) is a condition on a task that is satisfied when its *allow-continuation event* is fulfilled.

allow-continuation event The *allow-continuation event* into the standard specifications. Each task is attached an implicit *allow-continuation event* that is initially fulfilled.

omp_task_continuation_event The `omp_task_continuation_event` routine unfulfills and returns the *allow-continuation event* of the current task.

omp_fulfill_event(event) The `omp_fulfill_event(event)` routine can be extended as follows:

- **Constraints on Arguments** A program that calls this routine on an event that was already fulfilled is non-conforming. A program that calls this routine with an event handle that was not created by the detach clause, or not returned by the `omp_task_continuation_event` routine, is non-conforming.

The constraint on argument now also allows the task implicit continuation event to be passed to the routine. If the multiple clauses restriction on the `taskwait` construct were to be relieved as proposed, the `omp_fulfill_event(event)` could also be extended as follows:

- **Execution Model Events** The *task-fulfill* event occurs in a thread that executes an `omp_fulfill_event` region before the event is fulfilled if the OpenMP event object was created by a detach clause on a task or returned by the `omp_task_continuation_event` routine.

3.2 Implementation

We implemented the OpenMP extension in the Clang compiler, LLVM runtime and MPC runtime. First, we added a new ABI `__omp_taskwait_detach`, which fallbacks to the `omp_taskwait_detach` runtimes implementation. In both implementations the thread blocking schedule any other ready tasks until the passed event argument is fulfilled.

Then, we added support for the `detach(event)` clause to the `taskwait` construct in the Clang compiler: the `pragma omp taskwait detach(event)` directive will result in the new ABI being called.

4 Evaluation

We characterized three drawbacks on the `task detach(event)` approach and motivated how the `taskwait detach(event)` proposal mitigates these when suspending tasks during an asynchronous operation progression. This section provides additional evaluations on drawbacks (a) and (c).

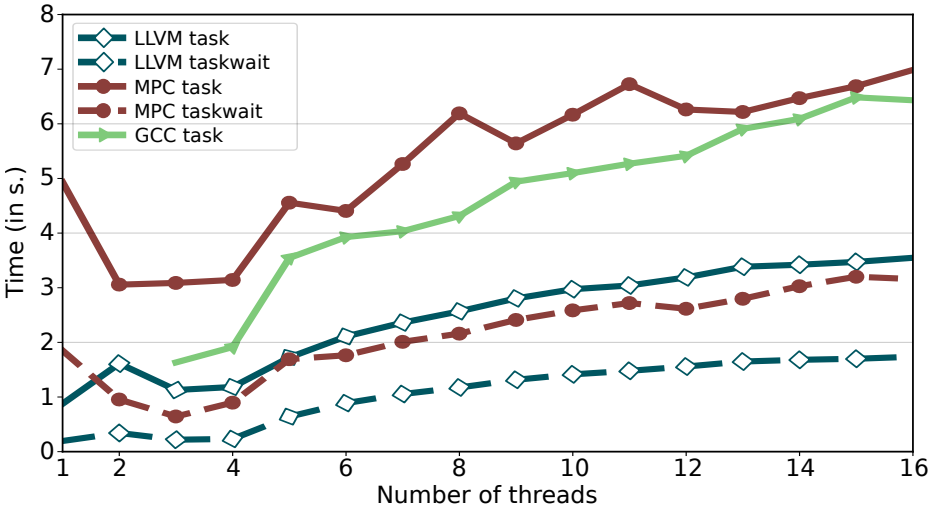

Fig. 1. Evaluation of the `task/taskwait detach` in LLVM, MPC and GCC with $N = 1,000,000$ in a NUMA region of an AMD EPYC CPU

4.1 Task Management Overheads

On (a), we made a microbenchmark to evaluate the tasking overheads of explicitly creating the continuation `task detach` against managing it implicitly using the `taskwait detach` proposal [9]. The microbenchmark has three-steps, it (1) warms up the runtime, (2) concurrently creates and schedules $2.N$ independent empty tasks with N empty dependent continuation tasks declared through the `task detach(event)` construct, and (3) concurrently creates and schedules $2.N$ independent empty tasks with an implicit sequential continuation task declared through the `taskwait detach(event)` construct we proposed. We tested on an AMD EPYC 7H12 CPU on a single NUMA region with $N = 1,000,000$ as a parameter, varying the number of threads to observe the impact of runtime contention. Fig. 1 presents the execution time of steps (2) and (3) for our patched LLVM 16.x and MPC implementations, and the execution time of step (2) for GCC 12.2.0 (median of 5 executions). The x-axis is the number of threads, and the y-axis is the execution time. No performances could be recorded for 1 and 2 threads on GCC as the runtime was deadlocking.

First, it seems LLVM runtime manages fine-grain tasks more efficiently than MPC and GCC. Though, we observe that both the LLVM and MPC `taskwait detach` implementation always outperforms the `task detach` implementation. As predicted in the mentioned in the motivations: the runtime manages twice as fewer tasks and no dependencies as opposed to the `task detach(event)` approach in this minimal example. We also observe that using more than 3

[9] `https://gitlab.inria.fr/ropereir/iwomp23/-/blob/main/bench/taskwait-detach.c`

threads deteriorates performances on every configuration as runtime contention on internal task data structure increases (tasks are empty).

In the LLVM implementation, using `taskwait detach` over `task detach` reduces per-task overheads for about $100ns$ depending on the number of threads. In applications such as Cholesky or LULESH, few tasks are suspended, so little there is little performance gain to expect from using our proposal over the `task detach(event)` solution. For instance on LULESH, tasks suspending could be those performing MPI communications. It only represents $\sim$ 20 tasks per iteration that may suspend, over $\sim 10,000$ computational tasks that may not suspend. However, our proposal could also ease application porting, which is the object of the two following sections.

4.2 Standardizing OpenMP Task Suspension on an Asynchronous Event

Currently, automatic task suspension when synchronizing on an external asynchronous event could be achieved as shown on Listing 1.4, using `pragmas` directly into the library implementation. However, OpenMP does not provide a standard ABI, and the (CUDA) runtime installation would be hardly dependant of a specific OpenMP runtime implementation.

```
1  int cuStreamSynchronize(CUstream hStream) {
2      if (/* within an OpenMP execution context */) {
3          omp_event_handle_t hdl;
4          # pragma omp task detach(hdl) depend(out: hdl)
5          {}
6          [...] /* differ stream synchronization to a progression engine */
7          # pragma omp taskwait depend(in: hdl)
8      } else {
9          [...] /* default implementation */
10     }
11 }
```

Listing 1.4. Runtime interoperability using task detach(event) approach

Our proposals provide a *standard* task suspension API/ABI. Using the proposed routines `omp_task_continuation_event` and `omp_taskwait_detach`, other asynchronous programming model runtimes (such as Cuda, MPI, libomptarget) could suspend OpenMP tasks and allow their continuation on an asynchronous event completion through a standard and portable interfaces across OpenMP implementations. This is depicted on Listing 1.5 on the CUDA and MPI runtime implementations; but could be extended to any other asynchronous programming model.

```
1  int cuStreamSynchronize(CUstream hStream) {
2      if (/* within an OpenMP execution context */) {
3          omp_event_handle_t hdl = omp_task_continuation_event();
4          [...] /* differ stream synchronization to a progression engine */
5          omp_taskwait_detach(hdl);
```

```
6       } else {
7           [...] /* default implementation */
8       }
9       [...]
10  }
11
12  int MPI_Send(...) {
13      if (/* within an OpenMP execution context */) {
14          omp_event_handle_t hdl = omp_task_continuation_event();
15          [...] /* differ request synchronization to a progression engine */
16          omp_taskwait_detach(hdl);
17      } else {
18          [...] /* default implementation */
19      }
20      [...]
21  }
22
23  # pragma omp task untied
24  {
25      [...]
26      cuStreamSynchronize(...);
27      [...]
28      MPI_Send(...);
29      [...]
30  }
```

Listing 1.5. Runtime interoperability using taskwait detach(event) API

4.3 Effort on Porting Existing MPI Applications

Scientific simulation codes are complex, and synchronizations (MPI, CUDA, I/O) may be hidden deep into libraries, as in the Arcane framework [4]. Automatic and standard interoperability would ease the porting of existing scientific applications to a task-based model, relieving programmers from the task suspension interoperability burden in such cases.

```
1   void CalcForceForNodes(Domain & domain) {
2       MPI_Irecv(comBuf, ..., recvRequest) ;
3       [...]    /* computation */
4       MPI_Wait(recvRequest, ...) ;
5       for (Index_t i = 0; i < opCount; ++i)
6           (domain.*dest)(dx*dy*(dz - 1) + i) += comBuf[i] ;
7   }
```

Listing 1.6. LULESH point-to-point communication pattern

The code Listing 1.6 corresponds to a simplified view of the MPI receive communications occurring in the LULESH [7] proxy-application. The function `CalcForceForNodes` function initiates non-blocking reception to temporary buffers (`comBuf`) on line 2. Then, it overlaps the reception with some computation

on mesh line 3. Finally, it waits for the reception completion line 4, and unpacks the temporary buffer to the mesh domain. A similar communication pattern can be found in most of the Collaboration of Oak Ridge, Argonne and Livermore (CORAL) proxy-applications. The continuation (line 4-7) is already explicit, and porting them with the `task detach` can be achieved by adding two tasks (line 2 and 4) as shown on Listing 1.7. Listing 1.8 shows a task-based version relying on runtime interoperability, that could be achieved using our proposal as discussed in the previous section. We move the continuation (message unpacking) right after a blocking `MPI_Recv` that shall suspend the task until the message is received, overlapping communication with other computational tasks. In such applications, our proposal reduces the runtime overheads (1 less task and dependency), and, likely, makes the code slightly more comprehensible: the continuation follows the predecessor sequentially, and synchronizations/overlap are managed implicitly by the task scheduler.

```
1  void CalcForceForNodes(Domain & domain) {
2      omp_event_handle_t event;
3      # pragma omp task detach(event) depend(out: recvRequest)
4      {
5          MPI_Irecv(comBuf, ..., recvRequest) ;
6          MPIX_Detach(&req, omp_fulfill_event, ev_handle);
7      }
8      [...] /* computation */
9      # pragma omp task depend(in: recvRequest)
10     {
11         for (Index_t i = 0; i < opCount; ++i)
12             (domain.*dest)(dx*dy*(dz - 1) + i) += comBuf[i] ;
13     }
14 }
```

Listing 1.7. LULESH task-based porting using task detach

```
1  void CalcForceForNodes(Domain & domain) {
2      # pragma omp task untied
3      {
4          MPI_Recv(comBuf, ...) ;
5          for (Index_t i = 0; i < opCount; ++i)
6              (domain.*dest)(dx*dy*(dz - 1) + i) += comBuf[i] ;
7      }
8      [...] /* computation */
9  }
```

Listing 1.8. LULESH task-based porting using runtime interfaces

5 Related Works

Weak Dependencies In cases presenting more concurrency than our minimal example, the `taskwait detach` proposal is not sufficient compared to the `task`

`detach`. In the example Listing 1.9, T3 depends on T1/T2, and T3/T4 cannot start until T2 executed and the event is fulfilled.

```
# pragma omp task depend(out: x) // T1
    [...]

# pragma omp task depend(out: y) detach(hdl) // T2
    [...]

# pragma omp task depend(in: x, y) // T3
    [...]

# pragma omp task depend(in: y) // T4
    [...]
```

Listing 1.9. Weak dependencies motivating example

In [17], authors introduce *weak* dependencies for OpenMP tasking. Weak dependencies could preserve both the concurrency and the task management costs saving from our proposal, as depicted on Listing 1.10. This code presents the same concurrency as the previous one, but T4 is executed as part of T2 following the C sequential order of execution, without the need of explicitly creating the continuation task T4 as previously.

```
# pragma omp task depend(out: x) // T1
    [...]

# pragma omp task depend(weakin: x) // T2
{
    [...]
    # pragma omp taskwait detach(hdl)

    # pragma omp task depend(in: x) // T3
        [...]

    [...] // T4
}
```

Listing 1.10. Weak dependencies example

Suspending OpenMP Tasks The current OpenMP specifications on `taskyield` construct, and `untied` clause are fully implementation-dependent. For instance, in GCC `untied` tasks will never change threads, and `taskyield` is implemented as a no-op but is yet fully standard-compliant. In LLVM, the compiler generates multiple *continuations* on each scheduling point appearing in the outermost scope of the task, which are then sent at run-time to the task scheduler in a continuation-passing style (CPS). It privatizes `untied` tasks outermost scope variables to ensure that the continuations restart in a coherent state. The `taskyield` construct is implemented as a continuation task if appearing in the outermost scope of the task; else, a new task is stacked on the current thread

(stack-algorithm presented in [22]). In the MPC-OMP runtime (implementing both GCC and LLVM ABI), the LLVM task continuations can be supported. In addition, tasks can be annotated to run on their own execution context (fixed-size stack + registers set). If suspending deep into the task call-stack, the task can be preempted for later resume on any thread without blocking onto its current thread. Every approach has its own limits. With GCC, scheduling flexibility is poor and may even lead to deadlocks when composing with other asynchronous programming models such as MPI, with threads spinning onto the same blocking task. With LLVM, scheduling points and variables must be fully known when compiling the task routine, which cannot be achieved for C Variable-Length Array (VLA) or Linux `alloca` routine leading to undefined behaviors [10]. With MPC-OMP, annotated tasks execution context may be created unnecessarily if the task never actually suspends; and statically fixing the stack size is an issue: too small, execution may stack-overflow; too big, unnecessary memory usage could impact performances. In [2], authors explore automated verification of stack size requirements for C programs at compile-time, that could complement MPC-OMP fixed-size stacks.

As a `taskwait` extension, our proposal integrates well into LLVM CPS tasking: the compiler could generate a continuation upon detecting a `taskwait detach(event)` construct on a task' outter-most scope. In other cases, stacking tasks frame on top of each other as done currently by LLVM/MPC restrict scheduling decision possibilities.

OpenMP as a Low-Level Parallel Runtime OpenMP is used as a low-level backend runtime for intra-node parallelization of higher-level programming models such as Kokkos [24], PGAS (XcalableMP [15]), or Domain Specific Languages/Abstraction (DSL/DSA) (Devito [11,13], Nablab [10]). They provide a higher abstraction that enables the user to write its code relieving programmers from low-level parallel implementation details. In the specific case of DSL using MLIR [9] compilers, OpenMP is easily targetable, and merging/suspending tasks can be achieved. Our contribution could be used by these programming models whenever a task must suspend.

Suspending Tasks in Rust The Rust programming language [8] supports asynchronous programming which, much like OpenMP, allows sequential-looking code to be run concurrently. A report of the Rust programming language in 2017 showed interest in CPS tasking as it is a lightweight solution that does not require per-tasks stack [11]. An experimental implementation was made the same year [12] to suspend tasks using `async/await` syntax. The authors mention that "*[Rust] coroutines are translated to state machines internally by the compiler*" which is similar to the LLVM `untied` tasks implementation. In addition, Rust

[10] https://github.com/llvm/llvm-project/issues/61499
[11] https://github.com/rust-lang/rfcs/blob/master/text/0230-remove-runtime.md
[12] https://github.com/rust-lang/rfcs/blob/master/text/2033-experimental-coroutines.md

is currently implementing a `Generator` type that is not yet in release builds of the language, to suspend tasks with the same continuation-passing style [13].

6 Conclusion

Many-core and heterogeneous architectures impose on users to compose multiple asynchronous programming models. OpenMP managing CPUs resources is responsible for orchestrating the suspension and progression of each asynchronous operation. In this paper, we proposed to extend the specifications by adding the `detach(event)` clause on the `taskwait` construct and defining the `omp_taskwait_detach(event)` API with a standard ABI. These extensions reduce programming and runtime overhead and are a step towards automating synchronizations overlap by OpenMP task scheduler when mixing asynchronous programming models.

In the future, we would like to evaluate OpenMP as an *asynchronous progression operation engine*: as depicted on Listing 1.5 line 4, remains the question of asynchronous progression responsibility. CUDA stream or MPI requests, asynchronous operations need CPU cycles at some point to progress. Most runtimes currently come with dedicated (p)threads blocking at the kernel level; another way is possible through cooperative task scheduling via OpenMP.

Acknowledgments

This preprint has not undergone peer review (when applicable) or any post-submission improvements or correction. The Version of Record of this contribution is published in IWOMP 2023 and is available online at https://doi.org/<DOI>

References

1. Bak, S., Bertoni, C., Boehm, S., Budiardja, R., Chapman, B.M., Doerfert, J., Eisenbach, M., Finkel, H., Hernandez, O., Huber, J., Iwasaki, S., Kale, V., Kent, P.R., Kwack, J., Lin, M., Luszczek, P., Luo, Y., Pham, B., Pophale, S., Ravikumar, K., Sarkar, V., Scogland, T., Tian, S., Yeung, P.: OpenMP application experiences: Porting to accelerated nodes. Parallel Computing **109**, 102856 (2022). `https://doi.org/10.1016/j.parco.2021.102856`
2. Carbonneaux, Q., Hoffmann, J., Ramananandro, T., Shao, Z.: End-to-End Verification of Stack-Space Bounds for C Programs. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 270–281. PLDI '14, Association for Computing Machinery, New York, NY, USA (2014). `https://doi.org/10.1145/2594291.2594301`
3. Ferat, M., Pereira, R., Roussel, A., Carribault, P., Steffenel, L.A., Gautier, T.: Enhancing MPI+OpenMP Task Based Applications for Heterogeneous Architectures with GPU Support. In: Klemm, M., de Supinski, B.R., Klinkenberg, J., Neth, B. (eds.) OpenMP in a Modern World: From Multi-device Support to Meta Programming. pp. 3–16. Springer International Publishing, Cham (2022)

[13] https://doc.rust-lang.org/std/ops/trait.Generator.html

4. Grospellier, G., Lelandais, B.: The Arcane Development Framework. In: Proceedings of the 8th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing. POOSC '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1595655.1595659
5. Iwasaki, S., Amer, A., Taura, K., Seo, S., Balaji, P.: BOLT: Optimizing OpenMP Parallel Regions with User-Level Threads. In: 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT). pp. 29–42 (2019). https://doi.org/10.1109/PACT.2019.00011
6. Kale, V., Lu, W., Curtis, A., Malik, A.M., Chapman, B., Hernandez, O.: Toward Supporting Multi-GPU Targets via Taskloop and User-Defined Schedules. In: Milfeld, K., de Supinski, B.R., Koesterke, L., Klinkenberg, J. (eds.) OpenMP: Portable Multi-Level Parallelism on Modern Systems. pp. 295–309. Springer International Publishing, Cham (2020)
7. Karlin, I.: LULESH Programming Model and Performance Ports Overview. Tech. rep. (12 2012). https://doi.org/10.2172/1059462
8. Klabnik, S., Nichols, C.: The Rust Programming Language. No Starch Press, USA (2018)
9. Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., Zinenko, O.: MLIR: Scaling compiler infrastructure for domain specific computation. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 2–14 (2021). https://doi.org/10.1109/CGO51591.2021.9370308
10. Lelandais, B., Oudot, M.P., Combemale, B.: Fostering Metamodels and Grammars within a Dedicated Environment for HPC: The NabLab Environment (Tool Demo). In: Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering. p. 200–204. SLE 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3276604.3276620
11. Louboutin, M., Lange, M., Luporini, F., Kukreja, N., Witte, P.A., Herrmann, F.J., Velesko, P., Gorman, G.J.: Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration. Geoscientific Model Development **12**(3), 1165–1187 (2019). https://doi.org/10.5194/gmd-12-1165-2019
12. Lu, H., Seo, S., Balaji, P.: MPI+ULT: Overlapping Communication and Computation with User-Level Threads. In: 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems. pp. 444–454 (2015). https://doi.org/10.1109/HPCC-CSS-ICESS.2015.82
13. Luporini, F., Louboutin, M., Lange, M., Kukreja, N., Witte, P., Hückelheim, J., Yount, C., Kelly, P.H.J., Herrmann, F.J., Gorman, G.J.: Architecture and Performance of Devito, a System for Automated Stencil Computation. ACM Trans. Math. Softw. **46**(1) (apr 2020). https://doi.org/10.1145/3374916
14. Meadows, L., Ishikawa, K.i.: OpenMP Tasking and MPI in a Lattice QCD Benchmark. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) Scaling OpenMP for Exascale Performance and Portability. pp. 77–91. Springer International Publishing, Cham (2017)
15. Murai, H., Nakao, M., Sato, M.: XcalableMP programming model and language. In: Sato, M. (ed.) XcalableMP PGAS Programming Language: From Programming Model to Applications, pp. 1–71. Springer Singapore. https://doi.org/10.1007/978-981-15-7683-6_1

16. Pereira, R., Roussel, A., Carribault, P., Gautier, T.: Communication-Aware Task Scheduling Strategy in Hybrid MPI+OpenMP Applications. In: IWOMP 2021 - 17th International Workshop on OpenMP. pp. 1–15. OpenMP : Enabling Massive Node-Level Parallelism (IWOMP 2021), Bristol, United Kingdom (Sep 2021). https://doi.org/10.1007/978-3-030-85262-7_14
17. Perez, J.M., Beltran, V., Labarta, J., Ayguadé, E.: Improving the integration of task nesting and dependencies in openmp. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 809–818 (2017). https://doi.org/10.1109/IPDPS.2017.69
18. Protze, J., Hermanns, M.A., Demiralp, A., Müller, M.S., Kuhlen, T.: Mpi detach - asynchronous local completion. In: Proceedings of the 27th European MPI Users' Group Meeting. p. 71–80. EuroMPI/USA '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3416315.3416323
19. Richard, J., Latu, G., Bigot, J., Gautier, T.: Fine-Grained MPI+OpenMP Plasma Simulations: Communication Overlap with Dependent Tasks. In: Yahyapour, R. (ed.) Euro-Par 2019: Parallel Processing. pp. 419–433. Springer International Publishing, Cham (2019)
20. Sala, K., Teruel, X., Perez, J.M., Peña, A.J., Beltran, V., Labarta, J.: Integrating blocking and non-blocking mpi primitives with task-based programming models. Parallel Computing **85**, 153–166 (2019). https://doi.org/https://doi.org/10.1016/j.parco.2018.12.008
21. Schuchart, J., Samfass, P., Niethammer, C., Gracia, J., Bosilca, G.: Callback-based completion notification using MPI Continuations. Parallel Computing **106**, 102793 (2021). https://doi.org/https://doi.org/10.1016/j.parco.2021.102793
22. Schuchart, J., Tsugane, K., Gracia, J., Sato, M.: The Impact of Taskyield on the Design of Tasks Communicating Through MPI. In: de Supinski, B.R., Valero-Lara, P., Martorell, X., Mateo Bellido, S., Labarta, J. (eds.) Evolving OpenMP for Evolving Architectures. pp. 3–17. Springer International Publishing, Cham (2018)
23. Tian, S., Doerfert, J., Chapman, B.: Concurrent Execution of Deferred OpenMP Target Tasks with Hidden Helper Threads. In: Chapman, B., Moreira, J. (eds.) Languages and Compilers for Parallel Computing. pp. 41–56. Springer International Publishing, Cham (2022)
24. Trott, C.R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D.S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., Wilke, J.: Kokkos 3: Programming Model Extensions for the Exascale Era. IEEE Transactions on Parallel and Distributed Systems **33**(4), 805–817 (2022). https://doi.org/10.1109/TPDS.2021.3097283
25. Véstias, M., Neto, H.: Trends of CPU, GPU and FPGA for high-performance computing. In: 2014 24th International Conference on Field Programmable Logic and Applications (FPL). pp. 1–6 (2014). https://doi.org/10.1109/FPL.2014.6927483