The Impact of Taskyield on the Design of Tasks Communicating Through MPI

Joseph Schuchart[1](✉), Keisuke Tsugane[2], José Gracia[1], and Mitsuhisa Sato[2]

[1] High-Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Stuttgart, Germany
{schuchart,gracia}@hlrs.de

[2] University of Tsukuba, Tsukuba, Japan
tsugane@hpcs.cs.tsukuba.ac.jp, msato@cs.tsukuba.ac.jp

Abstract. The OpenMP tasking directives promise to help expose a higher degree of concurrency to the runtime than traditional worksharing constructs, which is especially useful for irregular applications. In combination with process-based parallelization such as MPI, the `taskyield` construct in OpenMP can become a crucial aspect as it helps to hide communication latencies by allowing a thread to execute other tasks while completion of the communication operation is pending. Unfortunately, the OpenMP standard only provides little guarantees on the characteristics of the `taskyield` operation. In this paper, we explore different potential implementations of `taskyield` and present a portable black-box tool for detecting the actual implementation used in existing OpenMP compilers/runtimes. Furthermore, we discuss the impact of the different `taskyield` implementations on the task design of the communication-heavy Blocked Cholesky Factorization and the difference in performance that can be observed, which we found to be over 20 %.

Keywords: OpenMP tasks · Task-yield
Blocked Cholesky Factorization · Hybrid MPI/OpenMP · OmpSs

1 Motivation

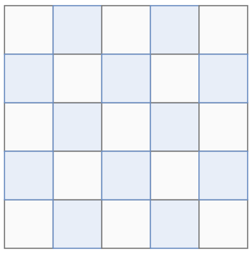
Task-based programming in OpenMP is gaining traction among developers of parallel applications, both for pure OpenMP and hybrid MPI + OpenMP applications, as it allows the user to expose a higher degree of concurrency to the scheduler than is possible using traditional work-sharing constructs such as parallel loops. By specifying data dependencies between tasks, the user can build a task-graph that is used by the scheduler to determine partial execution ordering of tasks. At the same time, OpenMP seems to become an interesting choice for higher-level abstractions to provide an easy transition path [9,10].

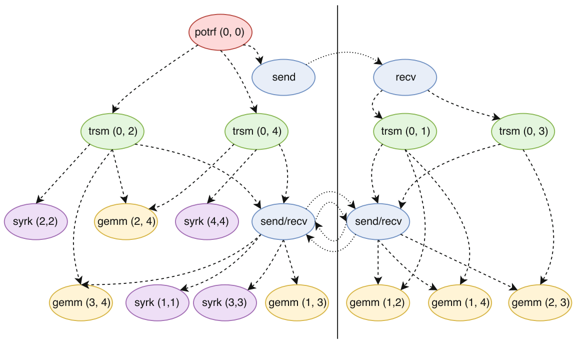
The OpenMP standard specifies the `taskyield` pragma that signals to the scheduler that the execution of the current task *may* be suspended and the thread may execute another task before returning to the current task (in case of `untied` tasks, the execution of the latter may be resumed by a different thread as well) [6]. This feature has the potential to hide latencies that may occur during the execution of a task, e.g., waiting for an external event or operation to complete. Among the sources for such latencies are I/O operations and (more specifically) inter-process communication such as MPI operations.

Traditionally, the combination of MPI + OpenMP using work-sharing constructs requires a fork-join model where thread-parallel processing interchanges with sequential parts that perform data exchange with other MPI ranks. Using OpenMP tasks, the user may structure the application in a way that communication can be performed concurrently with other tasks, e.g., perform computation that is independent of the communication operations [1,4]. In that case, communication may be split into multiple tasks to (i) parallelize the communication and (ii) reduce synchronization slack by achieving a fine-grained synchronization scheme across process boundaries, e.g., start the computation of a of subset of boundary cells as soon as all required halo cells from a specific neighbor have been received. The MPI standard supports thread-parallel communication for applications requesting the support for `MPI_THREAD_MULTIPLE` during initialization [5], which is supported by all major implementations by now.

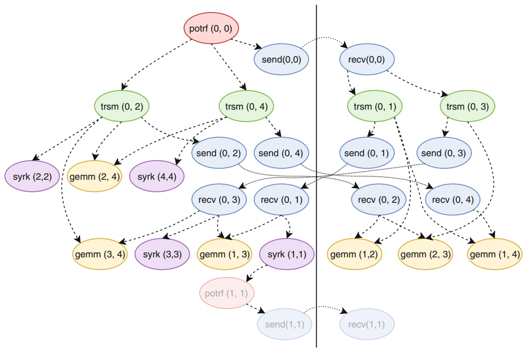
Fine-grained synchronization is especially useful for communication-heavy hybrid applications such as the distributed Blocked Cholesky Factorization, which decomposes a real-valued, positive-definite matrix A into its lower triangular matrix L and its transpose, i.e., $A = LL^T$. The computation of a block can naturally be modeled as a task that may start executing as soon as the blocks required as input are available, either coming from previous local tasks or received from another MPI rank. Figure 1 depicts two different approaches towards decomposing the computation and communication of blocks with a cyclic distribution across two processes (Fig. 1a) into tasks: Fig. 1b shows the use of coarse communication tasks, i.e., all send and receive operations are funneled through a single task. In Fig. 1c the communication is further divided into individual tasks for send and receive operations, which has the potential of exposing a higher degree of concurrency as tasks requiring remote blocks, e.g., some of the tasks at the bottom of the depicted graph, may start executing as soon as the respective receive operation has been completed. In particular, the synchronization stemming from the communication of blocks between tasks on different processes only involves a single producing task and a set of consuming tasks instead of including all producing and consuming tasks. While this may seem trivial for the depicted dual-process scenario, the difference can be expected to be more significant with increasing numbers of processes.

With MPI two-sided communication, a receive operation can only complete once a matching send operation has been posted on the sending side, and vice versa. At the same time, a portable OpenMP application should neither rely on the execution order of runnable tasks nor on the number of threads available to execute these tasks [6, Sect. 2.9.5]. Thus, it should not rely on a sufficient number of threads to be available to execute all available communication tasks, e.g., to make sure all send and receive operations are posted. In order to perform


(a) Cyclic block distribution



(b) Coarse-grain communication tasks



(c) Fine-grain communication tasks

Fig. 1. Task graph of the first iteration of the Blocked Cholesky Factorization on a matrix with cyclic distribution of 5×5 blocks across 2 processes with coarse- and fine-grain communication tasks. Local dependencies are depicted as strong dashed lines and communication as fine-dashed lines.

fine-grained synchronization using MPI two-sided communication, the user may be tempted to rely on `taskyield` to force the task scheduler to start the execution of all available tasks. Otherwise, the communication in Fig. 1c may deadlock as only either `send` or `recv` may be posted concurrently on both processes.

Unfortunately, the OpenMP standard is rather vague on the expected behavior of `taskyield` as it only specifies that the execution of the current task *may* be interrupted and replaced by another task. The standard does not *guarantee* that the current thread starts the execution of another runnable task (if available) upon encountering a `taskyield` directive. An implementation may safely ignore this directive and continue the execution of the currently active task. At the same time, OpenMP does not offer a way to query the underlying implementation for the characteristics of `taskyield`, making it hard for users to judge the extent to which fine-grained communication tasks may be possible.

In this paper, we discuss possible implementations of `taskyield` in Sect. 2. We further present a simple and portable black-box test to query the characteristics of `taskyield` and summarize the results on different OpenMP implementations in Sect. 3. Section 4 provides a performance comparison of different implementations of the distributed Blocked Cholesky Factorization that are adapted to the previously found characteristics. We draw our conclusions and outline future work in Sect. 5.

2 Potential Implementations of Taskyield

As mentioned above, the OpenMP standard leaves a high degree of freedom to implementations as to whether and how `taskyield` is supported. While this provides enough flexibility for implementations, it decreases the portability of application relying on specific properties of `taskyield`. In this section, we present a classification of possible implementations and outline some of their potential benefits and drawbacks.

No-op. Figure 2a depicts the simplest possible implementation: ignoring the `taskyield` statement simplifies the scheduling of tasks as a thread is only executing one task at a time without having to handle different task contexts (the context of a task is implicitely provided by the thread's private stack). For the purpose of hiding latencies, this implementation does not provide any benefit as tasks trying to hide latencies will simply block waiting for the operation to complete without any further useful work being performed in the meantime. Consequently, the <`slack`> of task T1 cannot be hidden and the runnable tasks T2 and T3 are only scheduled once T1 has finished its execution (① and ②). This may lead to deadlocks in two-sided MPI communication if not enough tasks are executed to post a sufficient number of communication operations.

Stack-Based. In Fig. 2b, a stack-based implementation of `taskyield` is depicted. Upon encountering a `taskyield` directive in task T1, the current thread retrieves the next runnable task T2 from the *Queue* (if any) and starts its execution on top of T1 (①). The execution of T1 resumes only after the execution of T2 finished. T1 may test the state of the operations it waits on and yield the thread to another task, e.g., T3 (②). This scheme provides a simple way for hiding latencies: tasks are not required to maintain their own execution context as again the stack of the current thread hosts the execution context and tasks are not enqueued into a queue upon encountering a `taskyield` directive. However, thread-private stacks are typically limited in size (albeit user-configurable) and thus implementations may have to limit the task-yield stack depth. Another drawback is the potential serialization of tasks: if all tasks that are being stacked upon each other happen to call `taskyield`, the re-entrance of the lower tasks is delayed until all tasks on top of them have finished their execution. Deadlocks in MPI two-sided communication may still occur if the number of operations is greater than the accumulated task-yield stack limit of the available threads.

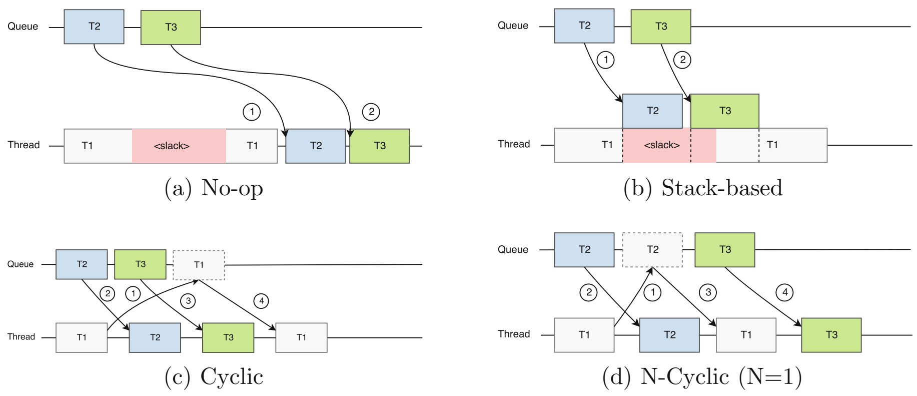


Fig. 2. Possible implementations of task-yield (single thread, three tasks).

Cyclic. An implementation supporting cyclic `taskyield` has the ability to reschedule tasks into the ready-queue for later execution by the same or another thread (in case the task is marked as `untied`). As depicted in Fig. 2c, upon encountering a `taskyield` (①) the thread moves the current task with its context to the end of the ready-queue and starts executing the next available task (②). Since the task T1 has been inserted at the end of the queue, its execution will only resume after the execution of T3 has completed or suspended through a call to `taskyield` (③ and ④). Provided that a sufficient number of runnable tasks is available, the latency of the operation started by T1 can be completely hidden behind the execution of T2 and T3. However, cyclic `taskyield` requires each task to maintain its own execution context as tasks are swapped in and out and thus cannot rely on the thread's stack to host their context. Even in the presence of a large number of communication tasks, cyclic `taskyield` guarantees the execution of all communicating tasks, thus avoiding deadlocks with MPI two-sided communication.

N-cyclic. The N-cyclic `taskyield` depicted in Fig. 2d is a generalization of the cyclic taskyield in that it inserts the yielding task at position N into the queue. This may be desireable by the user for two reasons: (1) a large number of tasks have been created and are runnable but the yielding task should resume its execution only after a few other tasks have been executed and before all of the runnable tasks executed, or (2) the available memory is scarce and may not be sufficient for all remaining tasks to allocate contexts and yield the thread, which would require a large number of contexts to be allocated at the same time. However, a hard-coded (small) limit N would introduce a similar potential for deadlocks in communication-heavy applications as the stack-based yield with limited stack-depth.

3 Existing Taskyield Implementations

We attempted to determine the actual variant of `taskyield` implemented by different OpenMP implementations. While some implementations are open source (GCC, Clang, OmpSs [3]), other implementations have to be treated as a black-box (Cray, PGI, Intel). Thus, we created a black-box test that tries to determine the characteristics of the underlying implementation.[1]

The test is presented in Listing 1.1. The main logic consists of a set of untied tasks with each task containing two `taskyield` regions (lines 15 and 34). The test logic relies on the constraint that only one thread is involved in its execution. Thus, all but the master thread are trapped (lines 9 through 11) in order to avoid restricting the enclosing parallel region to a single thread, which would otherwise skew the result.

It is important that the tasks are marked as `untied` as otherwise threads may not start the execution of sibling tasks, as mandated by the OpenMP task scheduling constraints [6, p. 94].

Before issuing the first task-yield, each task increments a shared variable and stores its value locally to determine the execution order (line 13). After returning from the first yield, the first task checks whether any other task has been executed in the meantime (line 19). If that is not the case, it is easy to determine that the `taskyield` was a `no-op` (line 20). Otherwise, the task continues to first check whether all or a subset of the remaining tasks have already executed the *second task-yield* and thus completed their execution, in which case the task-yield implementation is an unlimited (line 24) or depth-limited (line 26) `stack-based` implementation, respectively. If this is not the case, the task checks whether all tasks or a subset of tasks have at least reached the *first task-yield*, in which case we infer either a `cyclic` (line 28) or `N-cyclic` (line 30) task-yield.

Table 1 lists the implementations we detected using the black-box test discussed above. The result for GCC matches the expectations we had after examining the available source code of `libgomp`. In the case of both Clang and Intel, we found that more than one thread has to be requested to enable a stack-based yield with a depth limit of 257 tasks. Upon further investigation, it appears that this limit is imposed by the task creation throttling, i.e., the number of tasks the master thread creates before it starts participating in the task processing. We have not found a way to control the throttling behavior.

The Cray compiler behaves similarly with a stack-based yield limited by task creation throttling, although the imposed limit appears to be 97 tasks. For the PGI compiler, we determined a `no-op` task-yield.

The OmpSs compiler uses a task throttling mechanism by default, which imposes a configurable upper limit with a default of $T \times 500$ on the number of tasks active, with T being the number of threads [2]. Consequently, the task-yield should be considered `N-cyclic` with $N = T \times 500$. The throttling mechanism can be disabled, which leads to a `full-cyclic` task-yield.

[1] The full code is available at https://github.com/devreal/omp-taskyield.

Listing 1.1. Black-box test for `taskyield` implementations.

```c
  volatile int flag_one_cntr = 0;
  volatile int flag_two_cntr = 0;

#pragma omp parallel
#pragma omp master
  for (int i = 0; i < NUM_TASKS+omp_get_num_threads()-1; ++i) {
#pragma omp task firstprivate(i) untied
    {
      if (omp_get_thread_num() > 0) {
        // trap all but thread 0
        while(flag_one_cntr != NUM_TASKS) { }
      } else {
        int task_id = ++flag_one_cntr;

#pragma omp taskyield

        // when come back we only care about the first task
        if (task_id == 1) {
          if (flag_one_cntr == 1) {
            printf("NOOP\n");
          }
          // some other tasks were running in between
          else if (flag_two_cntr == (NUM_TASKS - 1)) {
            printf("STACK (unlimited)\n");
          } else if (flag_two_cntr == flag_one_cntr-1) {
            printf("STACK (depth=%d)\n", flag_one_cntr);
          } else if (flag_one_cntr == NUM_TASKS) {
            printf("CYCLIC\n");
          } else if (flag_one_cntr > 0) {
            printf("N-CYCLIC (N=%d)\n", flag_one_cntr-1);
          }
        }

#pragma omp taskyield

        ++flag_two_cntr;
      } // thread-trap
    } // pragma omp task
  } // for()
```

Table 1. Detected `taskyield` implementations using the black-box test depicted in Listing 1.1. T represents the number of threads.

Runtime	Version tested	Task-yield
GCC	7.1.0	No-op
Clang, Intel	Clang 5.0.1, Intel 18.0.1	No-op
	`OMP_NUM_THREADS > 1`	Stack (257)
Cray CCE	8.6.5	No-op
	`OMP_NUM_THREADS > 1`	Stack (97)
PGI	17.7	No-op
OmpSs	17.06, 18.04	Cyclic ($T \times 500$)
	`NX_ARGS="--throttle=dummy"`	Cyclic

Overall, our black-box test successfully detects the yield characteristics of existing OpenMP implementations. We hope that this test helps users experimenting with the OpenMP `taskyield` directive to choose the right task design depending on the OpenMP implementation at hand.

4 Evaluation Using Blocked Cholesky Factorization

To evaluate the impact of the different `taskyield` implementations on the performance of a communication-heavy hybrid application, we implemented several variants of the Blocked Cholesky Factorization.[2] The benchmark employs BLAS level 3 routines inside OpenMP tasks with defined input and output dependencies to create task graphs similar to the ones depicted in Fig. 1.

4.1 Implementations of Blocked Cholesky Factorization

Funneled Communication. We start from a version that funnels communication through a single task to exchange the blocks computed by the `trsm` tasks, as depicted in Fig. 1b. This version of the benchmark is guaranteed to work with all available OpenMP implementations and any number of threads, as no deadlock in the MPI communication may occur.

The single-task implementation comes in two flavors: in the `funneled` variant the communication task calls `taskyield` while waiting for the communication to finish whereas with `funneled-noyield` task-yield is not used. The latter serves as the baseline as it mimics the `no-op` task-yield on all OpenMP implementations.

Fine-Grained Communication Tasks. In contrast to `funneled`, the `fine` version of the benchmark creates a task per communication operation, leading to fine-grained task dependencies as depicted in Fig. 1c and a potentially higher degree of concurrency exposed to the scheduler. For this version, it is essential that a sufficient number of communication tasks can initiate block transfers to avoid starvation, e.g., $N + 1$ with N being either the number of receiving or sending tasks. Creating this version for `cyclic` task-yield only requires declaring input and output dependencies between the tasks, leaving it to the scheduler to properly execute them. As mentioned above, it is important that communication tasks are marked as `untied` as otherwise threads will not switch between tasks.

Unfortunately, employing fine-grained communication tasks with implementations offering stack-based task-yield requires more effort and has been an tedious process. Specifically, we were required to introduce *dummy tasks* to avoid communication tasks from different communication phases, i.e., tasks communicating the results of `potrf` and `trsm`, to overlap. This partial serialization of tasks is necessary as otherwise `recv` tasks (which naturally do not have local input dependencies) from a later communication phase may be scheduled on top of earlier `recv` tasks. The later `recv` tasks might then stall and never return to

[2] All code is available at https://github.com/devreal/cholesky_omptasks.

the earlier tasks due to implicit transitive dependencies through MPI between them. As an example, consider the `recv(1,1)` task depicted on the right in Fig. 1c being scheduled on top of `recv(0,0)`, which has the transitive implicit dependency `recv(0,0)` $\rightarrow$ `trsm(0,1)` $\rightarrow$ `send(0,1)` $\rightarrow$ `recv(0,1)` $\rightarrow$ `syrk(1,1)` $\rightarrow$ `potrf(1,1)` $\rightarrow$ `send(1,1)` $\rightarrow$ `recv(1,1)`. As a consequence, the user has to identify and expose these implicit dependencies stemming from two-sided communication, which OpenMP is otherwise not aware of.

Per-Rank Communication Tasks. We also made an attempt to reduce the number of communication tasks by combining all `send` and `recv` operations for a specific rank into a single task. This variant is called `perrank` and requires support for dependency iterators (as proposed in [7, p. 62]) to map dependencies from the block domain to the process domain, e.g., to collect dependencies on all blocks that have to be sent to a specific process. At the time of this writing, only OmpSs offered support for dependency iterators (called multi-dependencies).

Even with support for dependency iterators, the `perrank` version is not guaranteed to run successfully on `no-op` task-yield implementations as there are no formal guarantees on the execution order of tasks. Given a guaranteed similar relative execution order of the communication tasks on all processes, e.g., in the (reverse) order in which they were created, `perrank` will run successfully even on a single thread. However, if the execution order is random across processes, e.g., with a reverse order on some processes as a worst case, a minimum number of communication operations has to be in flight to avoid a deadlock. While we have not established a formal definition of this lower bound, we expect it to be below $\frac{(p-1)}{2}$, with p being the number of processes participating in the (potentially all-to-all) block exchange. However, the scalability in terms of processes may be limited by the number of threads available.

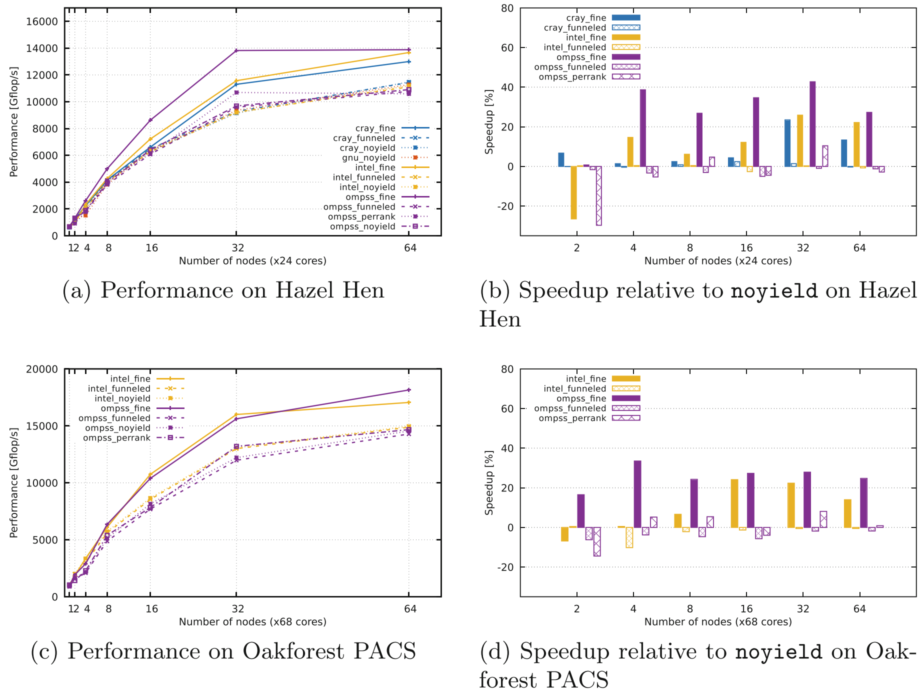
4.2 Test Environment

We ran our tests on two systems: *Oakforest PACS*, a KNL 7250-based system installed at the University of Tsukuba in Japan with 68-core nodes running at 1.4 GHz, and *Hazel Hen*, a Cray XC40 system installed at HLRS in Germany, which is equipped with dual-socket Intel Xeon CPU E5-2680 v3 nodes running at 2.5 GHz. On Oakforest PACS, we employed the Intel 18.0.1 compiler and Intel MPI 2018.1.163. On Hazel Hen, we used the Intel 18.0.1, GNU 7.2.0, and Cray CCE 8.6.5 compilers as well as Cray MPICH 7.7.0. On both systems, we used the OmpSs compiler and runtime libraries in version 17.06 in `ompss` mode.

On Oakforest PACS, we relied on the runtime system implementation to ensure proper thread pinning, i.e., using `KMP_AFFINITY=granularity=fine, balanced` for Intel. The OmpSs runtime performs thread pinning by default, which we had to explicitely disable on the Cray system as it would otherwise interfere with the thread pinning performed by `aprun`.

4.3 Results

Using the information from the black-box test presented in Sect. 3, we executed the different benchmark implementations discussed above with any suitable OpenMP compiler. We first ran the Cholesky factorization on matrices of size 64k^2 double precision floating point elements with a block size of 512^2 elements.



(a) Performance on Hazel Hen

(b) Speedup relative to `noyield` on Hazel Hen

(c) Performance on Oakforest PACS

(d) Speedup relative to `noyield` on Oakforest PACS

Fig. 3. Strong scaling performance and speedup relative to `noyield` of Blocked Cholesky Factorization on a 64k^2 matrix with block size 512^2 using different OpenMP compilers.

The measured performance of the benchmark on the Cray system is presented in Fig. 3a. The most important observation is that the implementations using fine-grained communication tasks (solid line) by far outperform the variant using funneled communication (dashed lines). The speedup of fine-grained communication tasks (depicted in Fig. 3b) range up to 42 % for OmpSs and up to 24 % for both Intel and Cray. For fine-grained dependencies, the OmpSs implementation outperforms the fine-grained implementations of the stack-based `taskyield` present in the Cray and Intel implementations. It is notable that OmpSs appears to saturate performance earlier – at 32 nodes – than the latter two implementations, which approach the saturation point at 64 nodes. It can also be observed that the difference between the `noyield` and the `funneled` version using `taskyield` is marginal, which can be attributed to the fact that yielding a single thread does not constitute a significant increase in resource availability given that a single thread occupies less than 5 % of the node's overall performance.

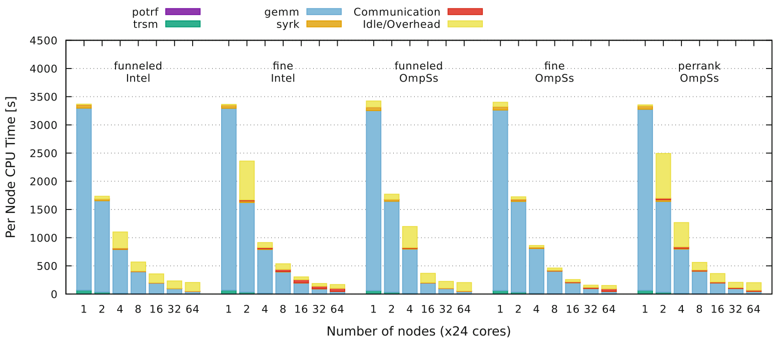
On Oakforest PACS, except for 64 nodes the different benchmark variants perform slightly better when using the Intel OpenMP implementation as compared to running under OmpSs, as depicted in Fig. 3c and d. This may be attributed to the generally higher overhead of task dependency handling observed in OmpSs [8]. However, relative to `noyield`, the versions using fine-grained communication tasks exhibit up to 34 % speedup with OmpSs and 25 % with Intel, with the main improvements seen in mid-range node numbers.

An interesting observation can be made on both systems regarding fine-grained communication tasks on the Intel runtime: for two nodes fine-grained communication tasks can have a negative impact on the performance, which diminishes and turns into performance improvements with increasing node numbers. This effect shrinks again for the highest node numbers.

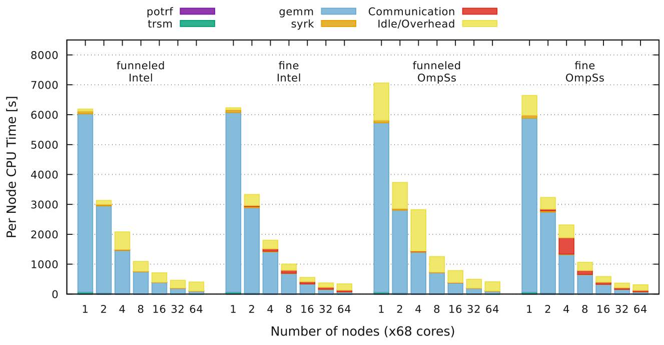
We also note that `perrank` communication tasks do not seem to provide significant benefits over `funneled` communication.

Breakdown of CPU Time. Figure 4 presents a breakdown of the accumulated time spent by the threads on the four BLAS operations and MPI communication as well as idle state and overhead. The latter contains task creation, task instantiation, task switching, and idle times as we have not found a way to further break down these numbers accurately. In all cases, the `gemm` kernel is the dominating factor but it is interesting to note that the overhead/idle times significantly increase with increasing node numbers for the `funneled` versions, indicating a lack of concurrency due to the coarse-grain synchronization. With the fine-grained synchronization, the idle times seem to be lower. However, at least for the Intel compiler the time spent waiting on MPI communication rises with increasing node numbers while this effect is less pronounced in OmpSs. We attribute this to the limiting effect on the available concurrency of stack-based task-yield: tasks below the currently executing task are blocked until the tasks above (and potentially the communication handled by them) have finished. In contrast to this, communication tasks in the `cyclic` task-yield in OmpSs finish as soon as the respective communication operations have finished and their execution has resumed once, allowing idle threads to poll for completion.

Scaling the Problem Size. Figure 5 depicts the strong scaling results of the Blocked Cholesky Factorization on a matrix with 128k^2 elements. Due to the computational complexity of $O(n^3)$, the total number of computation tasks increases by a factor of eight compared to a matrix size of 64k^2, leading to a total number of computation tasks of 2.8 million and putting significantly higher pressure on the task scheduler. On the Cray XC40 (Fig. 5a), OmpSs using fine-grained dependencies again outperforms all other implementations as it benefits from the larger number of available computational tasks, followed by the Intel compiler with fine-grained dependencies. All `funneled` runs show only limited

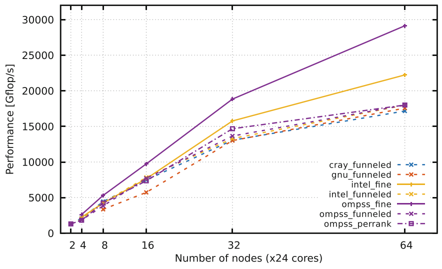


(a) Hazel Hen

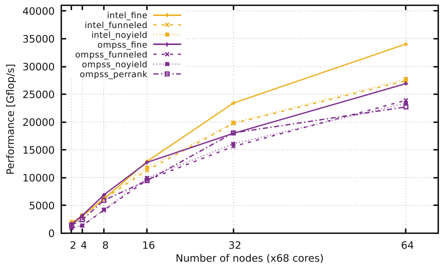


(b) Oakforest PACS

Fig. 4. Breakdown of CPU time of different implementations of the Blocked Cholesky Factorization for a 64k^2 matrix with block size 512^2.



(a) Hazel Hen



(b) Oakforest PACS

Fig. 5. Strong scaling Performance of Blocked Cholesky Factorization on a 128k^2 double precision floating point matrix with block size 512^2 using different OpenMP compilers.

scaling as they cannot exploit the full degree of concurrency present due to the coarse-grained synchronization. We should note that we were unable to gather reliable data for fine-grained communication tasks with the Cray compiler as we saw frequent deadlocks in the MPI communication, presumably due to the lower limit on the task-yield stack depth.

On Oakforest PACS (Fig. 5b), the scaling of OmpSs is rather limited compared to the Intel OpenMP implementation. We attribute these limitations to a relatively higher overhead involved in the OmpSs task management, which becomes more significant with the larger number of tasks and the low serial performance of a single KNL core. Again with both OmpSs and the Intel compiler, however, fine-grained communication tasks outperform the version using `funneled` communication on the same compiler. The `perrank` version appears perform slightly better, albeit with a far smaller benefit than the fine-grained communication tasks.

4.4 Discussion

The results presented above demonstrate that the available implementation of `taskyield` in OpenMP can have a significant impact on hybrid applications attempting to hide communication latencies, both in terms of task design, incl. correctness, and in terms of performance. While users can rely on a deadlock-free execution of communication tasks with `cyclic` task-yield, more care has to be given to the synchronization of tasks when using a `stack`-based or `N-cyclic` yield with fine-grained communication tasks. With both `no-op` yield and – less significant – stack-based yield the user has to ensure that a sufficient number of communication operations can be in-flight, e.g., by ensuring a sufficient number of OpenMP threads being available.

The variations of task-yield across OpenMP implementations make the transition from a correct, i.e., deadlock-free, sequential MPI application to a correct task-parallel MPI program tedious. In many cases, it might not be sufficient to simply encapsulate data exchange between individual computation tasks into communication tasks to achieve fine-grained synchronization and rely on OpenMP `taskyield` to ensure scheduling of all necessary communication tasks. Instead, users will have to work around the peculiarities of the different implementations and will thus likely fall back to funneling MPI communication through a single task to guarantee correct execution on all OpenMP implementations, potentially losing performance due to higher thread idle times.

However, our results strongly indicate that fine-grained communication tasks outperform more restricted synchronization schemes such as `funneled` and `perrank` as the former has the potential to significantly increase the concurrency exposed to the runtime. Unfortunately, an application has no way to query the current OpenMP implementation for information on the properties of `taskyield` to adapt the communication task pattern dynamically. Introducing a way to query these properties in OpenMP would allow users to adapt the behavior of their application to best exploit the hardware potential under any given OpenMP implementation. Similarly, providing a way to control the limits of task-yield in all implementations would help (i) raise awareness of the potential pitfalls, and (ii) provide the user with a way to adapt the runtime to the application's needs. Such configuration options could include the task creation throttling limit (as already offered by OmpSs) and any further limitation affecting the effectiveness of task-yield in the context of distributed-memory applications relying on two-sided communication.

5 Conclusion and Future Work

In this paper, we have presented a classification and evaluation of potential implementations of the `taskyield` construct in OpenMP. We discussed advantages and disadvantages of the different implementations and how a communication-heavy application may have to be adapted to successfully employ task-yield for communication latency hiding. Using a black-box test we were able to determine the characteristics of task-yield in different OpenMP implementations. We have shown that fine-grained communication tasks may outperform more coarse-grained approaches while requiring additional reasoning about implicit transitive dependencies from two-sided communication in `stack`-based task-yield implementations to avoid deadlocks.

Looking ahead, we plan to investigate other types of applications beyond Blocked Cholesky Factorization. While we do not expect the performance impact to be that pronounced on traditional stencil applications, applications from areas such as graph processing may benefit from taskified communication while potentially suffering from similar correctness problems with non-cyclic task-yield as presented in this paper. To help users tackle the issue of implicit transitive dependencies, it might be worth investigating ways to signal their existence to the OpenMP scheduler or provide users means to query or even control some scheduler characteristics, e.g., the relative task execution order and the properties of task-yield. From a user's (idealistic) perspective, the standard would eventually mandate a `cyclic` task-yield to avoid the issues described in this paper.

Acknowledgements. We gratefully acknowledge funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA) in the SmartDASH project. Part of this work has been supported by the European Community through the project Mont Blanc 3 (H2020 program, grant agreement N° 671697). This work was supported by a fellowship within the FITweltweit program of the German Academic Exchange Service (DAAD).

References

1. Akhmetova, D., Iakymchuk, R., Ekeberg, O., Laure, E.: Performance study of multithreaded MPI and OpenMP tasking in a large scientific code. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2017. https://doi.org/10.1109/IPDPSW.2017.128
2. BSC Programming Models: OmpSs User Guide, March 2018. https://pm.bsc.es/ompss-docs/user-guide/OmpSsUserGuide.pdf
3. Duran, A., et al.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. (2011). https://doi.org/10.1142/S0129626411000151

4. Meadows, L., Ishikawa, K.: 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.) IWOMP 2017. LNCS, vol. 10468, pp. 77–91. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_6
5. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (Version 3.1) (2015). http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
6. OpenMP Architecture Review Board: OpenMP Application Programming Interface, Version 4.5 (2015). http://www.openmp.org/mp-documents/openmp-4.5.pdf
7. OpenMP Architecture Review Board: OpenMP Technical report 6: Version 5.0 Preview 2 (2017). http://www.openmp.org/wp-content/uploads/openmp-TR6.pdf
8. Schuchart, J., Nachtmann, M., Gracia, J.: Patterns for OpenMP task data dependency overhead measurements. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2017. LNCS, vol. 10468, pp. 156–168. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_11
9. Tsugane, K., Lee, J., Murai, H., Sato, M.: Multi-tasking execution in PGAS language XcalableMP and communication optimization on many-core clusters. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region. HPC Asia 2018. ACM (2018). https://doi.org/10.1145/3149457.3154482
10. YarKhan, A., Kurzak, J., Luszczek, P., Dongarra, J.: Porting the PLASMA numerical library to the OpenMP standard. Int. J. Parallel Program. (2017). https://doi.org/10.1007/s10766-016-0441-6