© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI 10.1109/Cluster48925.2021.00024

# Combining One-Sided Communications with Task-Based Programming Models

Kevin Sala*, Sandra Macià† and Vicenç Beltran‡
*Barcelona Supercomputing Center (BSC)*
*Barcelona, Spain*
*Email: *kevin.sala@bsc.es, †sandra.macia@bsc.es, ‡vbeltran@bsc.es*

***Abstract*—Hybrid programming combining task-based and message-passing models is an increasingly popular technique to exploit multi-core clusters. The Task-Aware MPI (TAMPI) library integrates both models enabling the safe overlap of computation and communication tasks using two-sided MPI communications. Two-sided primitives combine data transfers with implicit synchronizations, but one-sided models usually offer more efficient data transfers decoupling synchronizations. MPI offers four distinct one-sided synchronization modes, while GASPI is a PGAS API providing one-sided operations with remote notifications for fine inter-process synchronizations.**

**In this paper, we study the challenges of integrating MPI and GASPI one-sided operations with the OpenMP and OmpSs-2 tasking models. We propose and implement several extensions to the GASPI and OmpSs-2 programming models, which are leveraged by a new library called Task-Aware GASPI (TAGASPI). The TAGASPI library allows the efficient and safe use of one-sided operations with remote notifications inside tasks. Both TAGASPI and TAMPI transparently manage communications issued by tasks and allow these to overlap with computation tasks naturally, following a data-flow model. These libraries are complementary and can be mixed in the same application.**

**Our experience porting several mini-apps to this hybrid model shows that TAGASPI helps leverage one-sided communications with similar complexity to pure and hybrid two-sided MPI approaches. We show that our hybrid one-sided approach outperforms the pure MPI strategies, but it also surpasses the TAMPI's performance when stressing communication phases, e.g., increasing the communication parallelism and reducing the communication tasks' sizes.**

***Keywords*-One-sided; RMA; Communication; Parallel programming; Message passing; MPI; GASPI; Data-flow; Task; OpenMP; OmpSs-2;**

## I. Introduction

The current trends of HPC system architecture show that present and future supercomputers will remain composed of thousands of compute nodes, and each node featuring an increasing amount of processing cores. Two key elements to achieve high scalability in such systems are communicating data between nodes from different cores simultaneously and overlapping computations and communications.

The Message Passing Interface (MPI) [1] is the main programming model in distributed environments. Most scientific applications use MPI to exploit both inter- and intra-node parallelism. However, MPI is not always the best choice for exploiting intra-node parallelism in modern multi-core processors. Pure MPI applications tend to be less flexible and more sensitive to load imbalance and differences in the performance of cores.

Hybrid parallel programming [2]–[4] is a traditional practice in the HPC community that mitigates the aforementioned issues. This technique combines distributed and shared-memory programming models, like MPI and OpenMP [5], to exploit inter- and intra-node parallelism, respectively. However, they are commonly combined following a simple but often suboptimal strategy called fork-join, where the computation phases are parallelized using OpenMP and MPI communications are mostly sequential. Some advanced techniques can be used to overlap computation and communication, like double buffering, but they usually increase the complexity of the application.

The hybrid technique that is currently gaining importance and does not increase the complexity is the taskification of both computation and communication phases without closing the parallelism in between. Tasks declare data dependencies on the data buffers that they are processing or communicating, defining a correct task execution order. This strategy follows a data-flow execution model that allows the natural overlap of computation and communication tasks. The Task-Aware MPI (TAMPI) library [6] [7] was recently proposed to safely and efficiently allow this taskification strategy for tasks that call two-sided MPI operations (e.g., `MPI_Recv` and `MPI_Isend`), whether blocking or non-blocking. With this library, computation and communication tasks can naturally overlap, while TAMPI transparently manages low-level communication aspects (e.g., the progress of MPI requests).

Currently, the TAMPI library only supports two-sided operations; thus one-sided communication primitives cannot be safely used inside tasks. Nevertheless, one-sided or RMA models allow processes to directly access the local memory of other remote processes with no intervention of the remote side. One-sided communication decouples data transfers from synchronization and can leverage RDMA hardware support, which is present in most modern network fabrics. These models minimize the overheads and complexities of two-sided interface implementations by removing the message tag matching, the message queuing, and the internal data buffering. Moreover, supporting multi-threaded communications efficiently in two-sided interfaces is more challenging than in one-sided.

The MPI standard defines one-sided operations to read and

write remote memory, along with multiple synchronization modes. However, it does not define a fine-grained mechanism to notify the remote side when the data has arrived on the remote memory. Several studies [8] [9] propose a lightweight notification mechanism, but it is not standardized yet. GASPI [10] defines a simple RMA interface with fine-grained remote notifications and can be mixed with MPI.

We aim at integrating the simple GASPI one-sided model with tasking programming models, such as OmpSs-2 [11]. To that end, we have developed a new library called Task-Aware GASPI that enables calling one-sided operations inside tasks safely and efficiently. Our proposal could also efficiently integrate one-sided MPI with tasking models once remote notifications [8] are included in MPI.

In this work, (1) we discuss the challenges of integrating MPI and GASPI one-sided operations with OpenMP and OmpSs-2 tasking models; (2) we design and implement the new library Task-Aware GASPI (TAGASPI) [12] that safely and efficiently supports one-sided GASPI operations inside OmpSs-2 tasks; (3) we extend GASPI with a fine-grained local completion waiting mechanism; (4) we study the interoperability between TAGASPI and TAMPI on the same application; (5) we extend OmpSs-2 to simplify the development of one-sided task-based applications; and (6) we perform an exhaustive performance evaluation of TAGASPI against two-sided MPI-only and TAMPI approaches.

## II. Background

### A. Remote Memory Access (RMA) in MPI

MPI has a one-sided RMA interface [1] [13] to access the local memory of other processes exposed through MPI windows. It defines the `MPI_Put` and `MPI_Get` operations to write/read data to/from another process' memory. MPI has two main one-sided synchronization modes. The *active* mode requires the receiver process to expose its memory explicitly before being accessed by other processes, whereas the *passive* mode makes all processes expose their memory permanently. Each mode is divided into two different sub-modes [13]. The *active* mode has the sub-mode with stronger synchronization, named *fence* sub-mode, where all processes in an MPI window synchronize with `MPI_Win_fence`. In contrast, the *passive* mode has the sub-mode with lower synchronization, called *global shared lock*, where processes can freely access the remote memories, and the user has to perform the required synchronizations to keep the memory consistency across processes. Ranks can transfer data using MPI RMA operations and use two-sided communication to notify the target rank about the completion of remote accesses [13]. MPI defines several synchronization methods for this sub-mode, such as `MPI_Win_flush` that waits for the remote completion of all RMA operations issued by the current process targeting a specific rank.

### B. GASPI One-Sided Interface

The GASPI standard [10] defines one-sided operations to write/read from other rank's memory (e.g., `gaspi_write`), exposed through memory segments similar to MPI windows. GASPI defines the `gaspi_write_notify` that writes and notifies the target process. The notification arrives at the remote side just after the data is written in the remote memory. This feature along with a function to wait for remote notifications constitute a fine-grained mechanism to notify and check the remote completion of one-sided operations. One-sided operations must be submitted to a specific communication queue. Queues are used to multiplex communications. Posting operations to the same queue and target rank guarantees the order of arrival at the remote side [10]. Lastly, GASPI defines the coarse-grained `gaspi_wait` to wait for the local completion of all RMA operations posted to a given queue, indicating when local communication buffers can be reused.

### C. Task-Aware MPI (TAMPI) Library

The MPI and OpenMP standards were not designed to efficiently perform MPI communications from multiple OpenMP tasks simultaneously [14] [6]. The Task-Aware MPI (TAMPI) library [6] [7] overcame this limitation by enabling the safe and efficient taskification of communications, allowing the overlap of computation and communication tasks that use blocking or non-blocking two-sided MPI operations (including both point-to-point and collectives). TAMPI supports non-blocking MPI operations by binding their MPI requests to tasks through the `TAMPI_Iwait` function. This non-blocking asynchronous function has the same parameters as the `MPI_Wait`. A task can call that function to bind its completion to the finalization of the MPI request passed as a parameter. Due to its asynchronous

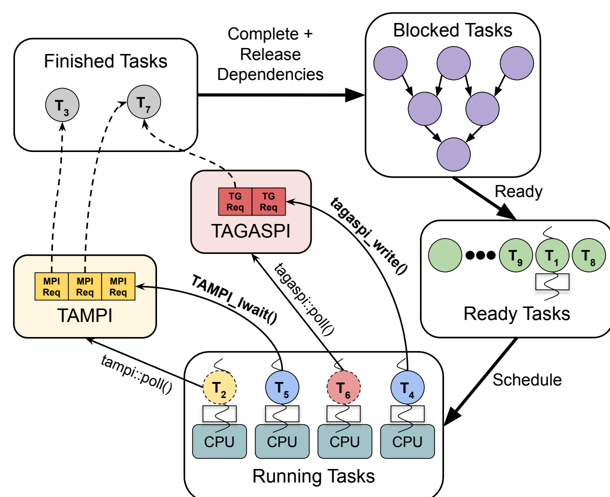


Figure 1. The flow of user tasks in a hybrid application with some tasks calling TAGASPI and TAMPI services. User tasks calling TAMPI and TAGASPI operations delay their completion (and release of dependencies) until their operations finalize. The polling tasks of these libraries are shown with dashed lines since they are transparent to the user.

nature, the function returns immediately without specifying whether the operation already finalized or not. The calling task continues but will not complete until (1) it finishes its execution and (2) all its bound MPI operations finalize. Figure 1 shows that behavior. In OpenMP and OmpSs-2, the data dependencies of a task are released just after the task completes. The non-blocking `TAMPI_Iwait` relies on the fact that the user annotates the communication tasks with the send buffers as input dependencies and the receive buffers as output dependencies. When one of these tasks completes, its dependencies on the communication buffers are released, and its successor tasks become ready to consume/reuse them.

The TAMPI library relies on the task external events API [6], present in OmpSs-2 [11], that allows tasks to bind their completion to the fulfillment of some external events. Figure 1 shows that tasks that finished the execution but have pending events (e.g., outstanding bound MPI requests) are kept in the finished state (gray tasks) until their events fulfill, when they complete and release their dependencies.

## III. Task-Aware RMA Communications

We aim at enabling a data-flow model that safely, efficiently, and effortlessly combines tasking models and one-sided operations. In this way, hybrid applications can fully taskify both computation and communication phases, connecting the tasks through the data dependencies on the buffers they are processing. Moreover, communication and computation tasks can naturally overlap, mainly thanks to the transparent scheduling and management of the tasking runtime system. But fine-grained and lightweight asynchronous RMA operations are the key elements to achieve efficient one-sided communication from multiple tasks in parallel.

The TAMPI library [6] already integrates two-sided MPI operations with tasking models. Thus, the first line of investigation is to explore an approach to make TAMPI support the RMA model offered by MPI. As explained in Section II-A, the MPI model defines four different synchronization modes for RMA communication. The first drawback to make MPI RMA operations task-aware is that all RMA synchronization functions are blocking. However, TAMPI relies on the non-blocking request-based variants of the MPI functions to implement its task-awareness. Moreover, this lack of non-blocking variants collides with the asynchronous philosophy of our proposal. This topic is not new; Zounmevo et al. [15] propose an MPI extension with non-blocking synchronizations. By standardizing these non-blocking variants, TAMPI could directly support all RMA modes.

However, this extension would not be enough to reach our objectives since most MPI RMA modes are very rigid and synchronous, hindering the efficient integration with tasking models. For instance, the *fence* mode requires opening/closing window epochs before/after performing RMA operations through that window, acting as parallelism barriers. Simple benchmarks can implement custom techniques mitigating those barriers, but it would be challenging in large applications with dynamic and irregular communication patterns.

The MPI RMA mode that better matches our objective is the *global shared lock* mode, which enforces less synchronization. However, the lack of a lightweight mechanism to notify the remote completion on the target side when performing an RMA operation is still a significant drawback. Commonly, the notification is implemented using a point-to-point communication between the sender and target rank [13].

```
MPI_Put(buffer, nelems, MPI_INT, dst, offset, nelems, MPI_INT, win);
MPI_Win_flush(dst, win);
MPI_Send(NULL, 0, MPI_INT, dst, tag, comm);
```

The code above exemplifies this notifying technique for the sender process, showing how the sender notifies that the data has arrived in the receiver's memory. The sender rank writes data on the receiver's memory using an `MPI_Put` and then flushes the window to wait for the remote completion of that RMA operation. After that, the sender knows that the data already arrived at the receiver side, so it sends an empty two-sided message. The receiver only has to wait for the message with a matching `MPI_Recv`. Notice that this notification could be an `MPI_Put` on a remote flag. Belli et al. [8] explain that this pattern requires an extra round-trip communication between the ranks during the flush. Internally, the flush operation requires the receiver to send an ack message to the sender indicating that the remote operation completed. Then, the sender sends an explicit MPI message back to the receiver indicating the same information. This extra round-trip communication and the MPI two-sided message for the notification undermine the use of RMA operations. Notice also that the flush is very coarse-grained because it waits for all the operations targeting a rank. The performance impact is negligible when the RMA messages are large enough but will become critical in our target scenario with fine-grained messages.

Belli et al. [8] propose an MPI extension to include a lightweight notification mechanism for efficiently solving this case. They present the `MPI_Put_notify` to write data to a target rank followed by a notification. This function requires an additional parameter indicating the tag that will use to notify the target rank. The receiver rank can register the waiting of the notification with the tag using a function that generates an MPI request. Then, it can wait for the notification arrival by calling `MPI_Wait` on that request.

By standardizing the non-blocking synchronizations [15] and remote notifications [8], TAMPI could provide a fine-grained RMA integration with tasking models without any significant change on the library. However, in this paper, we design and implement our approach based on the GASPI model [10] because it already provides all these properties. We can combine GASPI and MPI in hybrid applications to use GASPI for one-sided and MPI for two-sided primitives.

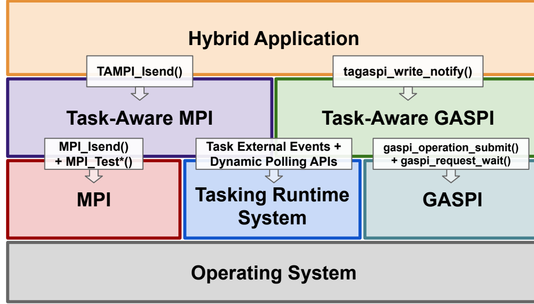


Figure 2. The software architecture for task-based hybrid applications that use our approach for one-sided and TAMPI for two-sided communications.

We materialize our proposal through a new external library called Task-Aware GASPI (TAGASPI) [12]. This library is inspired by the TAMPI's philosophy but targets the efficient and safe execution of one-sided operations inside tasks. Figure 2 shows the software architecture that we target for hybrid task-based applications, which can use one-sided (TAGASPI), two-sided (TAMPI) communications or both. These task-aware libraries are compatible with each other so that user applications can use their services simultaneously, even in the same communication tasks. Hybrid applications work on top of both task-aware libraries calling their public interfaces, although they can also call standard GASPI and MPI functions. In the background, TAGASPI and TAMPI communicate transparently with the GASPI and MPI libraries and use the services of the tasking runtime system.

## IV. Task-Awareness for GASPI

This section explains the design and implementation of the novel Task-Aware GASPI library, its common use-cases, and the extensions needed in the GASPI standard.

### A. *Task-Aware GASPI Library*

The Task-Aware GASPI (TAGASPI) is a new external library that targets the efficient and safe execution of one-sided hybrid GASPI+OmpSs-2 applications, maintaining the asynchronous philosophy of RMA communications. Our library allows tasks to perform fine-grained RMA operations and asynchronous waiting of remote notifications in parallel. Its services also simplify user applications' programmability compared to the significant changes needed to mix tasks and other RMA models. This library works above both GASPI and OmpSs-2 and transparently uses their APIs to orchestrate the task-awareness of communications.

The code of TAGASPI is publicly available in [12]. Application developers can link their applications to the TAGASPI library and call the API functions that it exposes, which we explain next. The library provides the `TAGASPI.h` header where declares all public methods and types. The functions `tagaspi_proc_init` and `tagaspi_proc_term` initialize and finalize both GASPI and TAGASPI libraries, respectively.

Our library provides a task-aware variant for each GASPI RMA operation, e.g., `tagaspi_write_notify`. Our variants,

```
1  int *A = (int *) malloc(N*sizeof(int)); // A is inside local_seg
2
3  #pragma oss task depend(in: A[0:N]) label(write data)
4  {
5      tagaspi_write_notify(local_seg, local_off,
6          dest, remote_seg, remote_off, N*sizeof(int),
7          /* Notification id & value */ 10, 1, queue);
8      // A[0;N] cannot be reused here!
9  }
10
11 #pragma oss task depend(inout: A[0:N]) label(reuse)
12 updateBuffer(A, N);
```

Figure 3. Example of using `tagaspi_write_notify` inside a task.

which are non-blocking and asynchronous, are intended to be called by tasks and behave similarly to the non-blocking `TAMPI_Iwait` [6]. A TAGASPI operation initiates the corresponding RMA operation, returning immediately but binding (and delaying) the completion of the calling task to the local finalization of the issued RMA operation. The task continues running, assuming that the operation might not be completed yet, and can finish the execution at any moment. Although it can finish executing, the task will not complete until its operations finalize. That delays the release of its data dependencies. Figure 1 shows how tasks calling TAGASPI operations delay their completion and release of dependencies until their operations finish. In the background, TAGASPI manages all in-flight RMA operations and periodically checks their finalization. Then, it transparently completes the finished tasks that had pending operations (in gray in Figure 1) once these operations end.

A communication task cannot assume that its issued TAGASPI operations have finished because that condition is guaranteed only once the task completes. Thus, the communication task itself cannot consume/reuse the communication buffers; only successor tasks can consume/reuse them safely. Users should annotate the tasks with the proper dependencies on the buffers that these are processing. For instance, to consume the data read remotely by a task that called `tagaspi_read`, we need this task to declare an output dependency on the local buffer (i.e., where the data will be stored) and another successor task declaring an input dependency on the same buffer acting as the consumer.

The same occurs to know when a local buffer can be safely reused after writing data from that buffer to remote memory. Figure 3 has an example of a task calling `tagaspi_write_notify`. The first task issues a *write+notify* to a remote rank taking buffer `A` as the data source (contained in segment `local_seg` at offset `local_off`), and a notification with id `10` and value `1`. The function returns right after issuing the RMA operation and binding the calling task's completion to the local finalization of the operation. Notice that the task declares buffer `A` as an input dependency (`A` is the data source). Thus, TAGASPI will transparently release that dependency once the task finishes its execution and the operation locally completes. Once released, the second task will become ready and will reuse buffer `A` eventually.

A task could issue multiple one-sided operations and may

```
1  int *B = (int *) malloc(N * sizeof(int)); // B corresponds to segment seg
2
3  #pragma oss task depend(out: B[0:N], notified) label(wait data)
4  tagaspi_notify_iwait(seg, 10, &notified);
5
6  #pragma oss task depend(in: B[0:N], notified) label(process)
7  processBuffer(B, N, &notified);
```

Figure 4. Example of using `tagaspi_notify_iwait` inside a task.

even perform unrelated computations in between. Notice we obsolete the standard `gaspi_wait` function (Section II-B) because now TAGASPI internally checks the local completion of task-aware RMA operations. In this way, we have removed the complexity of managing and waiting operations on communication queues. The application developers can focus on exposing the application's parallelism and are responsible just for deciding the queue where to post the operations.

Next, we explain the mechanism to wait for the completion on the target rank of an operation, named remote completion, which indicates the target rank that the operation has completed and the data is already on its local memory. The remote completion is especially useful in communication patterns where the target rank needs to know when the data arrives. As stated in Section II-B, GASPI guarantees that the notification of a *write+notify* will arrive at the remote side right after the data is written in the remote memory.

Our library offers the non-blocking function `tagaspi_notify_iwait` that initiates an asynchronous waiting of a notification. Only tasks should call this function, and its asynchronous behavior is similar to the one described for RMA previously. The completion of the calling task is delayed until the notification from the origin rank arrives (Figure 1). The function accepts a pointer parameter to where the notified value will be stored upon arrival.

Figure 4 shows an example of calling `tagaspi_notify_iwait` inside a task. The first task calls this new function to asynchronously wait for the arrival of the notification with identifier `10`. Upon arrival, TAGASPI will store the notified value in the `notified` variable. Note that the task declares an output dependency on this variable and another output dependency on buffer `B`. This code represents the receiver part that matches with the sender's code shown in Figure 3. The notification's arrival implies that the data sent by the sender rank (Figure 3) has already been written in the local buffer `B` (contained in segment `seg` in Figure 4). At that point, and if the first task has finished the execution, TAGASPI will complete the task and release its data dependencies. Then, the second task will become ready and will read and process the data received in buffer `B` eventually. The notification value, stored in `notified`, could be consumed too.

The task only waits for one remote notification in the previous example, but it could call `tagaspi_notify_iwait` multiple times to wait for different notification ids. The library also defines the function `tagaspi_notify_iwaitall` to wait for a consecutive range of notifications. Moreover, a task could mix calls of asynchronous waiting of notifications with the issuing of one-sided communications, such as `tagaspi_write`. The completion (and release of the dependencies) of such a task would be delayed until all these operations finish.

```
1  int *A = (int *) malloc(N*sizeof(int)); // A is inside local_seg
2
3  for (int i = 0; i < iterations; ++i) {
4      #pragma oss task depend(out: ack_notified) label(wait ack)
5      tagaspi_notify_iwait(local_seg, 20, &ack_notified);
6
7      #pragma oss task depend(in: A[0:N], ack_notified) label(write data)
8      tagaspi_write_notify(local_seg, local_off,
9          dest, remote_seg, remote_off, N*sizeof(int),
10         /* Notification id & value */ 10, 1, queue);
11
12     #pragma oss task depend(inout: A[0:N]) label(reuse)
13     updateBuffer(A, N);
14 }
```

Figure 5. Example of using `tagaspi_write_notify` inside a task protected by another task that waits an acknowledgment (ack) notification. This technique is useful in iterative producer-consumer patterns.

### *B. Supporting the Iterative Producer-Consumer Pattern*

Many HPC applications have an iterative producer-consumer communication pattern where a rank sends multiple data chunks to another remote rank in each iteration. Usually, the communication buffers on the receiver rank are the same across all iterations, so the target rank cannot receive the chunk of iteration `i` until it has consumed the same chunk received at the previous iteration `i-1`. Otherwise, the newer chunk would overwrite the older one before being consumed. Two-sided communication is not affected by this issue because the receiving of data does not start until the target process calls the receive function. However, in one-sided, the receiver side does not take part explicitly in the data transfer. Therefore, the sender must acknowledge when it is safe to write the data chunk on the receiver rank's memory. In TAGASPI, we solve this issue using a lightweight RMA notification to notify the sender rank when it can safely initiate the remote write. Throughout this text, we name this an ack notification.

We want to advance sending this ack notification to arrive at the sender side as soon as possible so that this latter can start the remote write without delays. The simplest way for the receiver is to send the ack notification inside the receiving task, which is the one waiting for the data chunk and its notification. However, this is suboptimal because we delay the sending of the ack notification and simulate a two-sided rendezvous communication. The optimal moment for sending the ack notification is inside the data consumer task, just after processing the data, where we know we will no longer use it. Supposing that the code in Figure 4 was inside a loop of iterations, we would place the call to `tagaspi_notify` after line 7 (inside the task). Note that the ack permits the sender to write the next iteration's chunk.

Figure 5 shows the code on the sender side. We have added a new task (`wait ack`) that declares an output de-

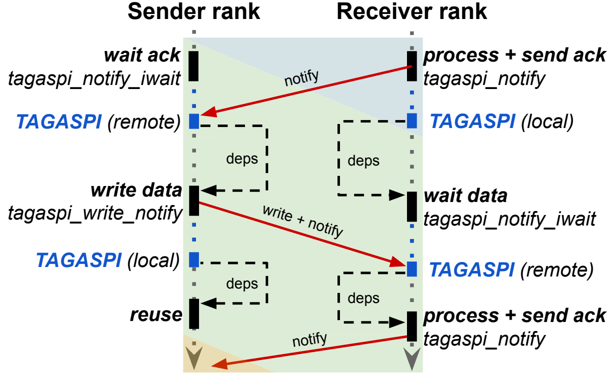


Figure 6. Vertical timeline of an iteration in an iterative producer-consumer pattern, where the sender rank (left) waits for the notification ack before writing data into the receiver's memory (right) using tasks and TAGASPI services.

pendency on the `ack_notified` variable and asynchronously waits for the ack notification sent by the receiver rank. The task that writes to the remote memory (`write data`) declares an input dependency also on `ack_notified`, so this latter task will not execute until the arrival of the ack. Figure 6 shows the execution timeline of one iteration in both processes. Black boxes represent the execution of tasks, connected through data dependencies inside each process (black dashed arrows), and some tasks communicating with the other process (red arrows). Blue boxes represent when TAGASPI, in the background, realizes that an issued RMA operation was completed locally (*local*) or whether a remote notification arrived (*remote*). This example uses a single data chunk, but real applications will work with multiple chunks in parallel. In Section V-A, we will show how to wait for an ack notification without the extra task.

### C. Fine-grained Local Completion in GASPI

The limitation that prevents implementing TAGASPI on top of the GASPI standard is that this latter does not provide a fine-grained wait mechanism for the local completion of issued RMA operations. Instead, it provides the coarse-grained `gaspi_wait` that waits for all the operations posted to a given queue. However, for our approach, we need (1) a mechanism to identify the RMA operations and (2) a new API function to wait for some operations posted to a queue and returning the identifiers of the completed ones.

We extend GASPI with a low-level API that exposes the underlying requests created by GASPI RMA operations but keeping the abstraction level. GPI-2 [16] divides each GASPI operation into various low-level requests, e.g., a *write+notify* uses two chained *ibverbs* requests. The requests of the *ibverbs* API have a 64-bit id which we can later retrieve when polling the completion of requests. Our idea is to allow issuing GASPI operations with an optional 64-bit tag and provide a new waiting API function returning the tags of the low-level requests completed upon the call.

We define a constant for each operation (e.g., `GASPI_OP_WRITE`) and a new API function `gaspi_operation_submit`. The function, shown below, submits any GASPI operation type, with a parameter for the operation type, the tag to identify all the low-level requests created by that operation, and all parameters that any operation may need (some may be ignored). Another new function, `gaspi_request_wait`, waits for the local completion of at most `max_reqs` low-level requests in the queue or until the timeout exceeds. It saves the number of completed requests in the output parameter and stores their tags and statuses (succeed or fail) in the output arrays.

```
gaspi_return_t
gaspi_operation_submit(gaspi_operation_t operation, gaspi_tag_t tag,
        /* All possible operation parameters... */);

gaspi_return_t
gaspi_request_wait(gaspi_queue_id_t queue, gaspi_number_t max_reqs,
        gaspi_number_t *completed_reqs, gaspi_tag_t array_of_tags[],
        gaspi_status_t array_of_statuses[], gaspi_timeout_t timeout);
```

### D. Implementation of the TAGASPI Library

Our library requires the tasking model to provide the external events API (Section II-C) to delay the completion of communication tasks by binding them events. We also use a mechanism to perform periodic checking of the pending RMA operations and the arrival of remote notifications. We have an internal task that periodically checks the completion of those operations (red dashed task in Figure 1). When one completes, it notifies the tasking system that the originating task fulfilled an event. This polling task is transparent to the application and runs every specific interval of time. In Section V-B we will show more details about this mechanism.

Implementing the RMA operations in TAGASPI, like `tagaspi_write_notify`, was straightforward. We show the internal code of this operation in Figure 7. Once an application task calls it, we retrieve the external event counter of the calling task (line 2). We increase the number of its events with the number of low-level requests that GASPI will create for that operation (line 3), delaying the task completion. The *write+notify* operation creates a low-level request for the write and another for the notify. Then, we submit the low-level GASPI operation by calling the new `gaspi_operation_submit` and passing a representative of the task as the operation tag (line 4). The representative is the opaque pointer provided by the tasking runtime system that identifies the task's event counter. After that call, the operation has been issued and the TAGASPI function returns, so the task can freely finish its execution. Then, in the background, the transparent polling task periodically checks all the active GASPI queues of the current process (line 7), looking for completed low-level requests using `gaspi_request_wait` in a non-blocking manner (line 13). We traverse the array of completed low-level requests (line 14) and decrease one external event from each event counter that was codified as the low-level operation tag (line 15).

```
1  gaspi_return_t tagaspi_write_notify(...) {
2    void *counter = get_current_event_counter();
3    increase_current_task_event_counter(counter, 2); // write + notify events
4    return gaspi_operation_submit(GASPI_OP_WRITE_NOTIFY, counter, ...);
5  }
6
7  void TAGASPI::pollRequests() { // called periodically by a polling task
8    gaspi_tag_t tags[MAX_REQS];
9    gaspi_status_t stats[MAX_REQS];
10   gaspi_number_t ncomp;
11
12   for (int q = 0; q < TAGASPI::_maxQueues; ++q) {
13     gaspi_request_wait(q, MAX_REQS, &ncomp, tags, stats, GASPI_TEST);
14     for (int c = 0; c < ncomp; ++c)
15       decrease_task_event_counter((void *) tags[c], 1);
16   }
17 }
```

Figure 7. Implementing `tagaspi_write_notify` inside TAGASPI.

The tasking system transparently checks if that was the last pending event of each task, and if so, it completes the task and releases its data dependencies, as shown in Figure 1.

Implementing `tagaspi_notify_iwait` was also quite simple. When a task calls this function, we first check whether the notification already arrived, and if so, we return directly without adding any external event. Otherwise, we get the current task's event counter, and we increase its events by one. Then, we get an auxiliary object to store the information about the pending notification: the notification id, the pointer to the location where to store the notified value, and the pointer to the event counter. This object must persist until the notification arrives, so we have a custom allocator managing a pool of these objects that uses a lock-free queue to keep the available ones. After initializing the object, we simply insert it into a lock-free multiple-producer-single-consumer (MPSC) queue holding all pending notification objects, and the TAGASPI function returns. In the background, the polling task also checks these pending notifications periodically. This task first extracts all the pending notification objects from the MPSC lock-free queue and inserts them into a Boost intrusive list (without dynamic allocations). Thus, the polling task can work with an efficient list, and the possible contention of the producers (communication tasks) does not affect the polling one. Recent studies [17] show lower overhead and contention in modern processors using this technique. The polling task checks (non-blocking) the arrival of each pending notification in the list. Once a notification arrives, we remove the notification object from the list and decrease the event counter of the task that is waiting for that notification. Again, the tasking system transparently completes the task and releases its dependencies if that was its last event.

## V. Adapting OmpSs-2 for Hybrid Applications

### A. Onready Action Clause

As explained in Section IV-B, many applications require the receiver rank to notify the sender when it is safe to write data to the receiver's memory. We showed our solution for these cases in Figure 5, where we had an additional task waiting asynchronously for the receiver's ack notification just before the writer task. However, this solution is not the most efficient for performance nor programmability, given that we are adding an extra task before every writer task.

```
1  void ack_iwait(gaspi_segment_id_t seg, gaspi_notification_id_t notid) {
2      tagaspi_notify_iwait(seg, notid, NULL);
3  }
4
5  for (int i = 0; i < iterations; ++i) {
6      #pragma oss task depend(in: A[0:N]) onready(ack_iwait(local_seg, 20))
7      tagaspi_write_notify(local_seg, local_off,
8          dest, remote_seg, remote_off, N*sizeof(int),
9          /* Notification id & value */ 10, 1, queue);
10
11     #pragma oss task inout(A[0;N]) label(reuse)
12     updateBuffer(A, N);
13 }
```

Figure 8. Example of using `tagaspi_write_notify` inside a task protected by an `onready` clause that waits an acknowledgment (ack) notification. This simplifies the sender code shown in Figure 5.

We propose a new task clause named `onready` that specifies a call to a user-defined function with any arguments. The tasking runtime will call this function only once, at any time after the task satisfies all its dependencies and before running its task body. The function cannot assume that it runs within a task context; it should not reach any task scheduling point. Nonetheless, it can register external events to the ready task to delay its execution until the events are fulfilled. For instance, the callback could execute TAGASPI operations or non-blocking TAMPI functions such as `tagaspi_notify_iwait` and `TAMPI_Iwait`. In this way, the data dependencies allow tasks to define local dependencies with other tasks on the same process, and the `onready` clause allows them to define remote dependencies with other processes (e.g., the arrival of a remote notification).

Figure 8 shows the new version of the sender code that we showed in Figure 5. We removed the additional task that waited for the ack notification (`wait ack`), and we moved that ack waiting inside an `onready` callback on the writer task, which is the one that writes data to the receiver's memory. In this way, once the communication buffer `A` is ready, the onready callback (`ack_iwait`) of the writer task will be executed. The callback will wait asynchronously for the ack notification, by calling `tagaspi_notify_iwait`, and will transparently register an event that will delay the execution of the writer task until the notification arrives. Upon arrival, the task will become ready and will run eventually. However, this would just be the worst-case scenario. Notice that if the receiver rank advances enough the ack notification, this latter could arrive much before the `onready` call, allowing the writer task to be scheduled without delays.

### B. Efficient and Dynamic Progress Checking

The task-aware libraries like TAGASPI require a service in the background checking the pending operations periodically. The existing TAMPI library has been using the polling services API [18] that allows registering a function that will be called periodically and opportunistically by the tasking

runtime. However, this polling API does not allow setting a polling frequency for each particular service. For this reason, we propose a new way of implementing the polling features. The idea is to have an isolated task inside TAGASPI scheduled periodically to check the pending operations. This task calls a new API function, named `wait_for_us` and provided by the tasking model, that blocks the calling task for a specific time in microseconds (`uint64_t` parameter). The task stops during that time approximately and yields the CPU to execute other tasks. The function returns the actual time slept so that TAGASPI could take decisions based on that. In TAGASPI, we spawn the polling task with the `nanos6_spawn_function` API function [11], which creates a task with an independent namespace of data dependencies and has no relationship with any other existing task.

This polling mechanism is very flexible because we can set a polling rate for each polling service. Although we did not implement it yet, the polling task could dynamically change the rate through the sleep time, e.g., depending on the number of pending operations. For a fair comparison, we have modified TAMPI to use the same mechanism to check the in-flight MPI requests (yellow dashed task in Figure 1).

## VI. Evaluation

In this section, we evaluate the performance and programmability of the Task-Aware GASPI library in three applications. In each one, we evaluate (1) an optimized two-sided MPI-only approach, (2) a two-sided hybrid MPI+OmpSs-2 variant that leverages TAMPI, and (3) a new one-sided hybrid GASPI+OmpSs-2 variant that leverages our novel TAGASPI library. In the MPI-only scenario, recent studies [8] show that the two-sided MPI strategies outperform standard one-sided MPI approaches (without extensions), besides two-sided strategies are much easier to implement. The objective of TAGASPI is to outperform MPI-only and TAMPI approaches in cases where communications are fine-grained and have an important weight. TAMPI and TAGASPI will show similar performance in many scenarios because TAMPI provides high scalability [6] [19] but will suffer higher contention (i.e., inside the MPI library) when many tasks communicate concurrently.

We run our experiments in the Marenostrum4 supercomputer with up to 256 nodes (12288 cores). Each node has two sockets Intel Xeon Platinum 8160 (2.10GHz) with 24 cores each (48 total cores), 96 GiB of memory, and an Intel Omni-Path HFI Silicon 100 Series network. We also use 16 nodes (1024 cores) of the CTE-AMD cluster. Each node has a single AMD EPYC 7742 (2.250GHz) with 64 cores (SMT is disabled), 1TiB of memory, and a Mellanox InfiniBand HDR100 network. We use the Intel 2017.4 compilers and Intel MPI 2017.4 on Marenostrum4, while Intel 2018.4 and OpenMPI 4.0.5 on CTE-AMD. We extended the GPI-2 v1.4.0 [16] with the modifications proposed in Section IV-C to support TAGASPI.

### A. Gauss–Seidel

We first use the iterative Gauss–Seidel method [20] that solves the Heat equation [21], a parabolic partial differential equation describing the heat distribution in a region over time. This benchmark derives from the Gauss–Seidel in the previous work [6] but using a 2–D matrix logically divided into blocks. The matrix is distributed across ranks assigning a consecutive set of rows to each one, so processes exchange the boundary rows with the upper and lower neighbors.

We evaluate an optimized MPI-only version [6] that uses non-blocking MPI primitives, advances their issuing as soon as possible, and waits for them when necessary, allowing the overlap of computation and communication. We also evaluate the hybrid MPI+OmpSs-2 variant that taskifies both computations and communications and leverages the TAMPI non-blocking support (`TAMPI_Iwait`). We develop a new hybrid version based on that taskification but using TAGASPI for communication. Sender tasks use `tagaspi_write_notify` to write and notify the receiver side, while receiver tasks just wait asynchronously for the remote notification calling `tagaspi_notify_iwait`. Sender tasks multiplex communications by posting operations through the multiple GASPI queues.

All variants work with a block size that controls the granularity of computations and communications. Since MPI-only spawns more processes, each one has a single row of blocks, so the block size specifies the columns per block. The hybrid variants have square blocks where the block size specifies their rows and columns, altering the task granularity.

We evaluate this benchmark in Marenostrum4; the MPI-only spawns 48 ranks/node, and the hybrid variants use one rank/socket (24 cores/rank) to avoid NUMA effects. Figure 9 shows the strong scaling experiment using the optimal block size of each variant. The upper figure shows the throughput's speedup, and the lower presents the parallel efficiency. We compute the speedup with respect to the throughput of the MPI-only variant in one node. We calculate the efficiency with respect to each variant's throughput in one node. Since the input is very large, we use a 16x smaller input from 1 to 8 nodes. We compute the speedup and efficiency based on the figure of merit (GUpdates/s).

The optimal block size is 1024 columns per block in MPI-only and blocks of 512x512 elements in the hybrid variants. The MPI-only performs slightly better with fewer nodes but ends performing worse at 128 and 256 nodes. The TAMPI version improves that performance in this latter scenario, but TAGASPI is the one scaling better. At 256 nodes, TAGASPI outperforms MPI-only and TAMPI by 1.15x and 1.06x, respectively. Although these results are already positive, this experiment works with optimal block sizes and communications do not have an important weight.

We run another experiment to observe how the variants behave when putting more pressure on communication.

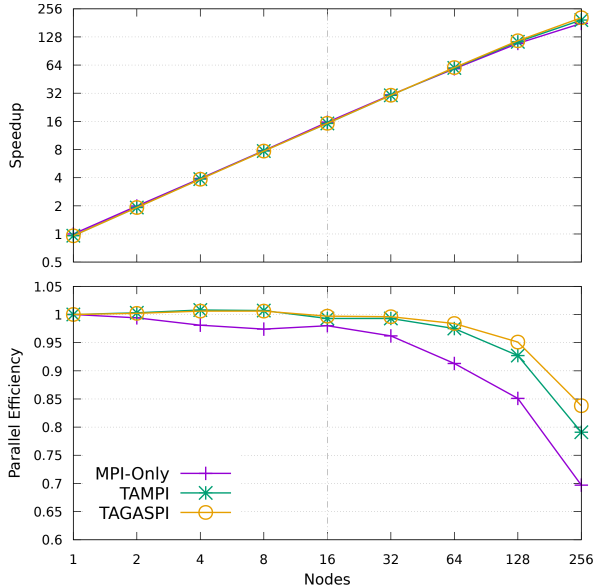

Figure 9. The Gauss–Seidel's strong scaling with a 256Kx128K matrix and 1000 timesteps in Marenostrum4 from 1 to 256 nodes. Due to the memory available in each node, we use a large input for the experiments from 16 to 256 nodes, and a 16x smaller input (64Kx32K matrix and 1000 timesteps) for the experiments from 1 to 8 nodes.

Figure 10 shows their throughput at 128 nodes halving the previous input and modifying the block size. Notice that changing the block size implies changing the computation and communication granularities, as well as the task granularity for the two hybrid versions. TAGASPI outperforms the rest in all cases, especially for configurations with a small block size, where the communication cost has a larger impact. The lower performance of TAMPI for small block sizes can be explained by the threading contention inside the MPI library, which we will discuss in the following sections. It is worth mention that the hybrid variants are also affected by the tasking overheads when running with small block sizes. However, for instance, TAGASPI using small blocks of 128x128 still works at an acceptable 60% of the peak throughput, while MPI-only is at 41% and TAMPI at 30%. That means that TAGASPI could scale to even more nodes with this same input using a small block size while keeping reasonable performance.

### *B. MiniAMR*

The second application is the miniAMR [22] [23], which mimics the communication, refinement, and load-balancing of larger adaptive mesh refinement applications. MiniAMR simulates the physics conditions of a 3–D domain when objects move across it. These objects create turbulent conditions in the regions they are present and miniAMR increases the simulation accuracy in those parts. The domain is initially divided into 3–D blocks and distributed among processes, but due to the dynamism of the simulation, turbulent blocks are refined into smaller blocks and redistributed periodically. MiniAMR features multiple phases of computation and communication interleaved, and then a refinement and load-balancing phase periodically. Previous works [24] [19] fully taskified its computation and communication phases and some parts of the refinement and load-balancing [19], using OmpSs-2 and TAMPI. We take that taskification as the base and port the communication phases to TAGASPI.

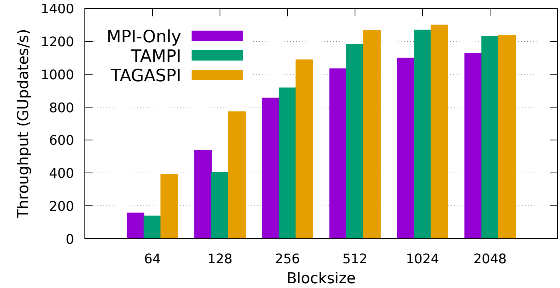

Figure 10. The Gauss–Seidel's throughput varying the block size with a 128Kx128K matrix and 500 timesteps in Marenostrum4 with 128 nodes.

Using RMA communication in miniAMR is not straightforward due to its dynamic and irregular communication pattern. Processes can have multiple neighbors, but they have only one memory buffer for sending and another for receiving, where block boundary data is packed/unpacked before/after the communication. Thus, it is difficult for a process to know where it should write the boundary block data in a neighbor process' memory. For this reason, we implement a sequential phase just after the refinement and load-balancing stages where each pair of neighboring processes agree on the unique remote offset and notification identifier for each of the RMA messages that they will exchange during regular communication phases. In this way, sender tasks know precisely where they should write their data on the remote side, whereas the receiver tasks wait for the corresponding notifications. The communication phase follows an iterative producer-consumer pattern (Section IV-B). The receiver side (consumer) must permit any sender task (producer) before this latter writes to its receiving buffer, preventing the overwrite of the data sent in the previous iteration. We solve this issue by making the receiver side send an ack notification once the data has been unpacked from the receiver buffer. In turn, the sender tasks asynchronously wait for the corresponding ack using the `onready` clause (Section V-A). Although we need the extra agreement phase after the refinement, the complexity of the communication phases keeps very similar to the TAMPI variant, where simple non-blocking MPI sends and receives are used. Both TAGASPI services and the `onready` clause strongly help keep the programmability and readability of the code when using RMA communication.

We run the following experiments in Marenostrum4; the MPI-only uses 48 ranks/node, and hybrids use 4 ranks/node and 12 cores/rank. That is the optimal configuration for hybrid approaches in miniAMR, given that the refinement

phase is not fully taskified [19]. In the TAGASPI variant, we also use TAMPI during the load-balancing stage to demonstrate that both libraries can work together. The load-balancing stage represents a small portion of the total time and does not present improvement opportunities, so we keep this stage with tasks that call two-sided TAMPI services.

Firstly, we perform a strong scaling experiment with the same input used in the previous study of miniAMR [19], but we halve the number of computed variables (to 20 variables) to reduce the computational weight. The hybrid variants send/receive/write each boundary block face from a different task (separate messages). That is not the optimal configuration (the optimal is around eight faces per message) but provides very reasonable performance [19] and puts more pressure on the communication phases. We show the throughput speedup of the strong scaling on the upper part of Figure 11 and the parallel efficiency on the lower part. We compute the speedup with respect to the throughput of the MPI-only variant in one node. We calculate the efficiency with respect to each variant's throughput in one node. Again, since the input is very large, we use a 16x smaller input from 1 to 8 nodes. We compute the speedup and efficiency based on the figure of merit (GUpdates/s). We show the results for the whole algorithm time and the results assuming negligible refinement time (marked as NR). The NR results are the ones we would see in an ideal execution with the refinement taking negligible time to run. These are useful to observe the impact of the refinement phases [19].

In this case, TAGASPI achieves the best scalability and efficiency; it improves both MPI-only and TAMPI by 1.41x at 256 nodes. The efficiency of TAGASPI is significantly better since it ends with an efficiency of 0.84, while MPI-only is at 0.73 and TAMPI at 0.58 (non-refinement). As expected, the throughput assuming negligible refinement time is significantly better in all variants because the refinement has several sequential sections. Notice that TAMPI scales well up to 32 nodes, but it starts decreasing the efficiency from 64 nodes due to the high pressure on the communication. In those cases, TAMPI would need to increase the communication granularity to mitigate that effect.

We perform another experiment with 128 nodes using the previous input but varying the computed variables from 10 to 40 to see the impact on each variant. Figure 12 shows the throughput of each variant in this experiment. Again, TAGASPI performs better in all configurations with significant differences. MPI-only is barely affected by the number of variables but has lower performance. The hybrid versions computing 10 variables show low throughput because the small granularity of computation tasks brings up the tasking runtime's overheads. TAMPI improves as we increase the computed variables and reduce the pressure on the communication side. The largest differences are with 20 variables, where TAGASPI outperforms MPI-only and TAMPI by 1.46x and 1.40x (non-refinement), respectively.

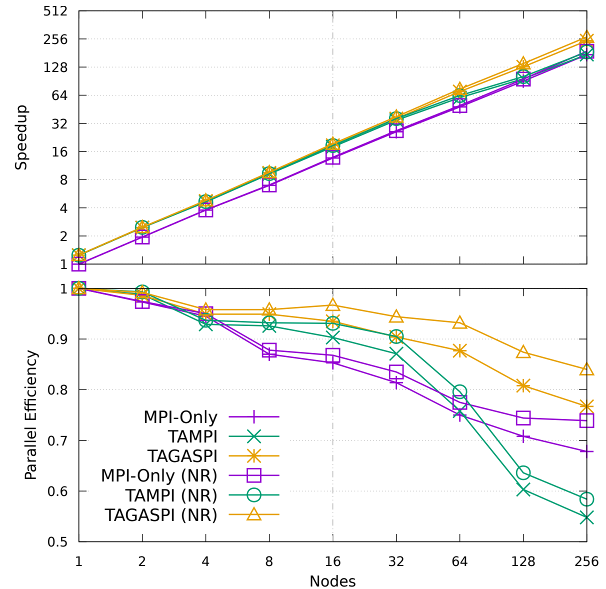


Figure 11. The miniAMR's strong scaling in Marenostrum4 from 1 to 256 nodes. The lower shows the efficiency for both the total time and assuming a negligible refinement time (NR). Due to the memory available in each node, we use a large input for the experiments from 16 to 256 nodes, and a 16x smaller input for the experiments from 1 to 8 nodes.

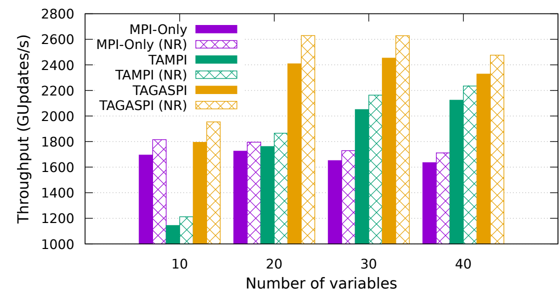


Figure 12. The miniAMR's throughput varying the computed variables in Marenostrum4 with 128 nodes. The figure shows the throughput for both the total time and assuming a negligible refinement time (NR). Notice that the vertical axis (throughput) starts at 1000 GUpdates/s.

During an analysis of the execution traces of the hybrid variants with 20 variables, we observed that the tasks using TAGASPI communications are much faster than with TAMPI. For instance, the sender and receiver tasks using TAGASPI are around 5x and 100x faster than the TAMPI ones, respectively. This difference is mainly explained by the high contention inside the MPI library when calling `MPI_Isend` and `MPI_Irecv` concurrently from several tasks. In contrast, the GASPI model allows GPI-2 [16] to implement communications with lower threading synchronization. We will explain more details about the contention inside the MPI libraries in the following application's section.

### C. *Streaming*

The last application is a new communication-intensive benchmark that we call Streaming, inspired by the Pipelined

Stencil application [8]. This benchmark processes multiple large chunks of data, and each participating compute node has a different function that must apply to each chunk. There are no data dependencies between elements of a chunk when applying a function. From the first node, a chunk moves across all the rest of the nodes, one by one, applying each function to the chunk's elements. We repeat this process for each chunk, one after the other, following a pipeline of chunk computations across the nodes. We can specify a block size, which is the granularity of computations and communications, as well as the granularity of tasks in the hybrid variants. Blocks of a chunk can be processed by a node concurrently. The computational weight and communication pattern are the same independently from the configuration of ranks per node. Each process receives/sends data from/to a different node. They have a receive and a send buffer with sufficient space to receive/send all the blocks of a single chunk simultaneously. We have an optimized MPI-only variant using non-blocking operations, a hybrid taskified variant with non-blocking TAMPI, and our hybrid TAGASPI variant. The communication has an iterative producer-consumer pattern, so the TAGASPI variant requires ack notifications and uses the `onready` clause on writer tasks.

We evaluate the Streaming on Marenostrum4, using 48 ranks/node for MPI-only and two ranks/node (24 cores/rank) in hybrid runs. The upper part of Figure 13 shows the throughput of each variant when varying the block size with 64 nodes. We also show the standard deviation in each point as error bars. In this case, the MPI-only is the variant that performs better in general, but TAGASPI almost reaches it when using blocks of 2K or more. TAMPI achieves its peak performance with 8K block size but has significantly worse performance than TAGASPI with smaller block sizes. TAGASPI does not outperform MPI-only because Intel MPI is specially optimized for the Intel Omni-Path networks and the PSM2 API. In contrast, the *ibverbs* API (leveraged by GPI-2 [16]) is emulated in this fabric.

We demonstrate that TAGASPI can outperform MPI-only by running on the CTE-AMD cluster that has an InfiniBand network. We show the throughput with 16 nodes in the lower part of Figure 13. MPI-only uses 64 ranks/node, and the hybrid variants use one rank/node (64 cores/rank). In this case, TAGASPI significantly outperforms both TAMPI and MPI-only. For instance, with blocks of 4K, TAGASPI improves MPI-only and TAMPI by 1.53x and 2.14x, respectively. Moreover, the MPI-only variant suffers from more considerable variability in this system.

The TAGASPI variant significantly outperforms TAMPI's in small and medium block sizes, as in the previous applications. These differences are mainly because of the contention inside the MPI library when communicating in parallel from several tasks. Implementing an efficient locking strategy for the `MPI_THREAD_MULTIPLE` level in the MPI libraries is challenging, although there have been many improvements

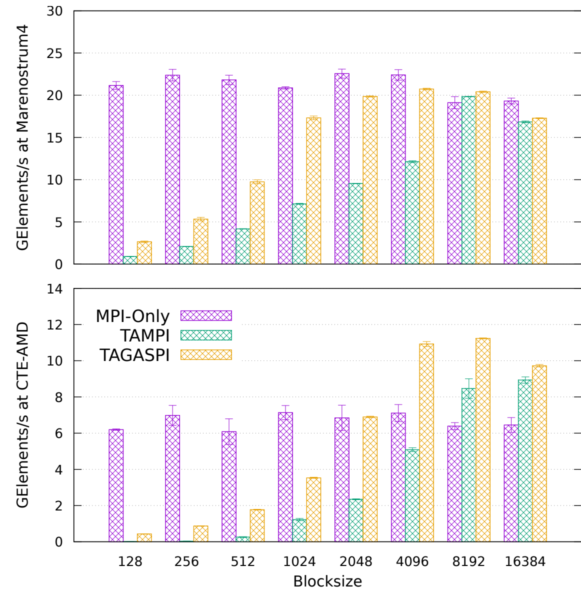


Figure 13. The Streaming's throughput varying the block size in Marenostrum4 with 64 nodes (upper) and CTE-AMD with 16 nodes (lower). The upper computes 250 chunks of 768K elements each and the lower with 250 chunks of 1024K elements.

during the last years. We analyze why the TAMPI variant performs poorly with small block sizes by comparing the time spent inside MPI using different block sizes in Marenostrum4. The TAMPI version almost reaches the peak throughput with block size 8192 and drops nearly to half with 2048. In this case, we have observed that the total time inside MPI (among all threads of an MPI rank) with block size 2048 increases up to 27x the MPI time with block size 8192. An analysis with the Intel VTune profiler has reported that the wait time inside the MPI library's locking system increases enormously when using the small block size. Specifically, the wait time is dedicated to acquiring a lock shared between the most time-consuming MPI functions called by this application (`MPI_Isend` and `MPI_Irecv`) and TAMPI internally (`MPI_Test` and `MPI_Testsome`).

Leveraging TAMPI or TAGASPI requires setting the frequency in which their transparent polling task checks the completion of their communications. The optimal value of this frequency depends on the communication intensity of the user application, the communication software stack, and the HPC system hardware. In Marenostrum4, the optimal values for the frequency of the TAMPI and TAGASPI polling task are running every $150 \mu s$ for Gauss–Seidel and miniAMR, and $50 \mu s$ for Streaming. In CTE-AMD, Streaming needs the TAGASPI task to run every $50 \mu s$, and TAMPI needs a dedicated core ($0 \mu s$). Streaming requires more frequency due to its high communication intensity.

Lastly, we want to mention that our Task-Aware GASPI library is being used in Saiph [25] [26], which is a Domain-Specific Language (DSL) that facilitates the simulation of physical phenomena from the Computational Fluid Dynam-

ics (CFD) domain in HPC systems. Macià et al. [27] extend Saiph with a new back-end that generates a hybrid task-based GASPI+OmpSs-2 variant of a high-level application and internally leverages the TAGASPI library to perform one-sided communications.

## VII. Related Work

We designed our task-aware features based on one-sided operations with lightweight notifications provided by GASPI. The concept of remote notifications is widely used in other PGAS and RMA models. For instance, some PGAS models like the Fortran CoArrays [28] are implemented using a similar concept. Belli et al. [8] propose extending the one-sided MPI interface with notifications, as well as Sergent et al. [9]. Recent studies [29] describe the lack of remote notifications in MPI as one of its major drawbacks. Once the MPI standard extends one-sided operations with notifications, we will be able to apply our task-aware features to fine-grained one-sided MPI communication with competitive performance.

Some studies optimize applications using hybrid GASPI and OpenMP approaches. The common strategy is to issue one-sided operations concurrently and wait for their completion from the master thread. The next computation phase starts once all operations complete. This fork-join approach is usually suboptimal since the synchronization point after the parallel region limits both the intra- and inter-node parallelism. Thébault et al. [30] use the fork-join strategy to parallelize irregular applications with tasks. Möller et al. [31] also mix GASPI and OpenMP tasks with a similar strategy in an electromagnetic simulation application. However, to the best of our knowledge, no studies efficiently integrate tasks with GASPI enabling a true data-flow execution model.

Schuchart et al. [32] present the concept of global dependencies among tasks that run on different processes and based on the DASH PGAS model [33]. Tasks can declare both local and remote dependencies so that inter-process synchronizations can be defined as dependencies among tasks at different processes. Our approach allows implementing more fine-grained inter-process synchronizations and complex communication patterns and requires much less effort from developers when porting their applications to our approach. The HabaneroUPC++ [34] integrates the Habanero-C *async-finish* tasking model with the UPC++ [35] PGAS model. This model has the previous drawbacks, but it also treats each communication as a task, which could bring overhead issues due to an excessively fine-grained tasking. This latter issue is also present in HCMPI [36], which integrates Habanero-C tasks with MPI, but it does not support one-sided operations.

## VIII. Conclusion and Future Work

In this paper, we have discussed the challenges of integrating the one-sided interfaces of MPI and GASPI with tasking models. We have designed and implemented the Task-Aware GASPI library to efficiently support GASPI operations called by OmpSs-2 tasks in a fine-grained way. We have extended both GASPI and OmpSs-2 to implement our library and allow programmability enhancements in one-sided applications. We show significant performance improvements against MPI-only and hybrid TAMPI approaches, such as a 1.40x speedup in miniAMR. As future work, we expect the remote notifications (or similar) to be standardized in MPI so that we can efficiently integrate one-sided MPI with tasks through the TAMPI library. We also plan to investigate about dynamically adapting the running frequency of the TAGASPI polling task based on the workload, e.g., the number of outstanding operations.

## Acknowledgment

We thank Dr. Valeria Bartsch, Dr. Daniel Grünewald and Dr. Mirko Rahn (Fraunhofer ITWM, Germany) for the early discussion of the required changes in the GASPI interface and the GPI-2 implementation to support the Task-Aware GASPI library. This work has been supported by the European Union H2020 Programme through the LoSync PRACE-6IP project (agreement No. INFRAEDI-823767); the Spanish Ministry of Economy through the Severo Ochoa Center of Excellence Program (SEV-2015-0493); the Spanish Ministry of Science and Innovation (PID2019-107255GB); and the Generalitat de Catalunya (2017-SGR-1414).

## References

[1] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard. Version 3.1*. University of Tennessee, Jun. 2015.

[2] R. Rabenseifner and G. Wellein, “Comparison of Parallel Programming Models on Clusters of SMP Nodes,” in *Proceedings of the International Conference on High Performance Scientifc Computing*, Mar. 2003.

[3] R. Rabenseifner, “Hybrid parallel programming: Performance problems and chances,” in *45th Cray User Group Conference*, 2003, pp. 12–16.

[4] G. Jost, H. Jin, and F. F. Hatay, “Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster,” NASA, Tech. Rep., Sep. 2003.

[5] OpenMP Architecture Review Board, “OpenMP Technical Report 8: Version 5.1 Preview,” November 2010, Accessed on: 2021-07-28. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-TR8.pdf

[6] K. Sala, X. Teruel, J. M. Perez, A. J. Peña, V. Beltran, and J. Labarta, “Integrating blocking and non-blocking MPI primitives with task-based programming models,” *Parallel Computing*, vol. 85, pp. 153–166, 2019.

[7] Barcelona Supercomputing Center, “Task-Aware MPI (TAMPI) Library,” Accessed on: 2021-07-28. [Online]. Available: https://github.com/bsc-pm/tampi

[8] R. Belli and T. Hoefler, “Notified access: Extending remote memory access programming models for producer-consumer synchronization,” in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 871–881.

[9] M. Sergent, C. T. Aitkaci, P. Lemarinier, and G. Papauré, “Efficient notifications for mpi one-sided applications,” in *Proceedings of the 26th European MPI Users’ Group Meeting*, 2019, pp. 1–10.

[10] GASPI Forum, “GASPI: Global Address Space Programming Interface. Version 17.1,” February 7th 2017, Accessed on: 2021-07-28. [Online]. Available: http://www.gaspi.de/gaspi/

[11] Barcelona Supercomputing Center, “OmpSs-2 Specification,” Accessed on: 2021-07-28. [Online]. Available: https://pm.bsc.es/ompss-2-docs/spec/

[12] ——, “Task-Aware GASPI (TAGASPI) Library,” Accessed on: 2021-07-28. [Online]. Available: https://github.com/bsc-pm/tagaspi

[13] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood, “Remote memory access programming in mpi-3,” *ACM Transactions on Parallel Computing (TOPC)*, vol. 2, no. 2, pp. 1–26, 2015.

[14] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, “Overlapping communication and computation by using a hybrid MPI/SMPSs approach,” in *Proceedings of the 24th ACM International Conference on Supercomputing*, 2010, pp. 5–16.

[15] J. A. Zounmevo, X. Zhao, P. Balaji, W. Gropp, and A. Afsahi, “Nonblocking epochs in mpi one-sided communication,” in *SC’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 475–486.

[16] Fraunhofer ITWM, “GPI-2,” Accessed on: 2021-07-28. [Online]. Available: https://github.com/cc-hpc-itwm/GPI-2

[17] D. Álvarez, K. Sala, M. Maroñas, A. Roca, and V. Beltran, “Advanced synchronization techniques for task-based runtime systems,” in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 334–347.

[18] K. Sala, J. Bellón, P. Farré, X. Teruel, J. M. Perez, A. J. Peña, D. Holmes, V. Beltran, and J. Labarta, “Improving the interoperability between MPI and task-based programming models,” in *Proceedings of the 25th European MPI Users’ Group Meeting*. ACM, 2018, p. 6.

[19] K. Sala, A. Rico, and V. Beltran, “Towards data-flow parallelization for adaptive mesh refinement applications,” in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020, pp. 314–325.

[20] A. Greenbaum, *Iterative Methods for Solving Linear Systems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997.

[21] G. Esposito, “The heat equation,” in *From Ordinary to Partial Differential Equations*. Springer, 2017, pp. 329–334.

[22] C. T. Vaughan and R. F. Barrett, “Enabling tractable exploration of the performance of adaptive mesh refinement,” in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 746–752.

[23] C. T. Vaughan, “MiniAMR Adaptive Mesh Refinement (AMR) Mini-app,” Accessed on: 2021-07-28. [Online]. Available: https://github.com/Mantevo/miniAMR

[24] A. Rico, I. S. Barrera, J. A. Joao, J. Randall, M. Casas, and M. Moretó, “On the benefits of tasking with OpenMP,” in *International Workshop on OpenMP*. Springer, 2019, pp. 217–230.

[25] S. Macià, S. Mateo, P. J. Martínez-Ferrer, V. Beltran, D. Mira, and E. Ayguadé, “Saiph: Towards a dsl for high-performance computational fluid dynamics,” in *Proceedings of the Real World Domain Specific Languages Workshop*, 2018, pp. 1–10.

[26] S. Macià, P. J. Martínez-Ferrer, S. Mateo, V. Beltran, and E. Ayguadé, “Assembling a high-productivity dsl for computational fluid dynamics,” in *Proceedings of the Platform for Advanced Scientific Computing Conference*, 2019, pp. 1–11.

[27] S. Macia, K. Sala, and V. Beltran, “Leveraging the Task-Aware GASPI’s One-Sided Communications in Saiph,” Jul. 2021, Technical Report. [Online]. Available: https://doi.org/10.5281/zenodo.5144803

[28] A. Fanfarillo and D. D. Vento, “Notified access in coarray fortran,” in *Proceedings of the 24th European MPI Users’ Group Meeting*, 2017, pp. 1–7.

[29] F. Liu, C. Barthels, S. Blanas, H. Kimura, and G. Swart, “Beyond mpi: New communication interfaces for database systems and data-intensive applications,” *ACM SIGMOD Record*, vol. 49, no. 4, pp. 12–17, 2021.

[30] L. Thébault and E. Petit, “Asynchronous and multithreaded communications on irregular applications using vectorized divide and conquer approach,” *Journal of Parallel and Distributed Computing*, vol. 114, pp. 16–27, 2018.

[31] N. Möller, E. Petit, Q. Carayol, Q. Dinh, and W. Jalby, “Scalable fast multipole method for electromagnetic simulations,” in *International Conference on Computational Science*. Springer, 2019, pp. 663–676.

[32] J. Schuchart and J. Gracia, “Global task data-dependencies in pgas applications,” in *International Conference on High Performance Computing*. Springer, 2019, pp. 312–329.

[33] K. Fürlinger, T. Fuchs, and R. Kowalewski, “Dash: A c++ pgas library for distributed data structures and parallel algorithms,” in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. Ieee, 2016, pp. 983–990.

[34] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar, “Habaneroupc++ a compiler-free pgas library,” in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, 2014, pp. 1–10.

[35] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "Upc++: a pgas extension for c++," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 1105–1114.

[36] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with MPI," in *27th International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2013, pp. 712–725.