Extending MPI One-Sided Communication with Lightweight Notification-Based Synchronization

Arnau Cinca
Barcelona Supercomputing Center
Barcelona, Spain
arnau.cinca@bsc.es

Kevin Sala
Barcelona Supercomputing Center
Barcelona, Spain
kevin.sala@bsc.es

Kiril Dichev†
Independent Researcher
Zurich, Switzerland
kiril.dichev@gmail.com

Albert-Jan Yzelman †
Independent Researcher
Zurich, Switzerland
albert-jan@yzelman.net

Vicenç Beltran
Barcelona Supercomputing Center
Barcelona, Spain
vbeltran@bsc.es

***Abstract*—As high-performance computing systems scale toward many-core architectures, hybrid programming models combining message passing and task-based execution require efficient communication and synchronization. However, integrating communication and synchronization into these models remains challenging.**

We present two contributions to address these challenges. First, we introduce Task-Aware LPF (TALPF), a task-integrated implementation of the Lightweight Parallel Foundations library supporting asynchronous communication and relaxed synchronization while preserving the analyzable Bulk Synchronous Parallel model and enabling overlap of computation and communication. Second, we extend a notification-based MPI communication model to support multiple occurrences of the same notification identifier, enabling fine-grained synchronization without additional overhead. This mechanism is exported through Task-Aware MPI (TAMPI) and underpins Two-Sided RDMA (2SR), which emulates two-sided MPI semantics using one-sided operations with notifications.

We evaluate these contributions on an HPC system and demonstrate competitive or superior performance compared to state-of-the-art task-aware and MPI one-sided implementations while improving programmability, safety, and portability.

***Index Terms*—HPC, Many-Core Systems, Task-based Programming Models, Bulk Synchronous Parallel, One-Sided Communication Models, OmpSs-2, MPI, LPF, GASPI**

I. Introduction

Over the past decade, the end of Dennard scaling [10], the rise of dark-silicon constraints [11], and power density limitations have driven high-performance computing (HPC) architectures toward increasing core counts rather than higher clock frequencies. As a result, modern compute nodes now expose hundreds of hardware threads, with exascale systems projected to scale to thousands. In this context, traditional flat Single Program, Multiple Data (SPMD) deployment—where one process is bound to each core and processes interact among them using communication libraries such as MPI [17] or GASPI [24]—begin to show their limitations [3]. Duplicated halo regions inflate memory usage, collective operations incur overheads proportional to the number of ranks, and even minor load imbalances or system noise can ripple across ranks, significantly degrading overall efficiency.

To address these challenges, hybrid programming models have become increasingly common. In such models, a coarse-grained set of processes handles inter-node communication, while intra-node parallelism is exploited through shared-memory and threads. Although many hybrid applications still follow the traditional fork-join structure, recent advancements have accelerated the adoption of task-based runtimes. The emergence of task-aware (TA) communication libraries such as Task-Aware MPI (TAMPI) [21, 22] and Task-Aware GASPI (TAGASPI) [19], allows both computation and communication to be expressed as a directed acyclic graph (DAG) of tasks, which can be dynamically scheduled by the runtime system. This data-flow execution model allows the system to transparently overlap communication and computation, eliminating the need for complex and error-prone manual orchestration.

As a more structured alternative to these dynamic paradigms, the Bulk-Synchronous Parallel (BSP) model [26] remains appealing due to its simplicity and predictability. BSP-style execution consists of alternating computation, communication, and synchronization phases, a pattern that is widely used implicitly, particularly in large-scale scientific applications (where communication may itself act as a synchronization primitive, e.g., `MPI_Recv`). Each BSP superstep comprises a local computation phase followed by a communication phase, and a process may only begin the next superstep once all its incoming and outgoing communication has completed. This rigid synchronization requirement is realised explicitly by libraries such as Lightweight Parallel Foundations (LPF) library [25] and BSPlib [14] with LPF enforcing the model in a particularly strict manner. While this structure simplifies reasoning about correctness and performance, it introduces high synchronization overhead and limits compatibility with task-based runtimes (e.g., OpenMP, OmpSs-2). These limitations stem from the model's rigid global synchronization, which hinders overlap across supersteps—precisely the fine-

† This work was done while affiliated with the Computing Systems Lab, Huawei Research Zurich.

© 2026 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.

This is the accepted version of: A. Cinca, K. Sala, K. Dichev, A. N. Yzelman, and V. Beltran, "Extending MPI One-Sided Communication with Lightweight Notification-Based Synchronization," in *ExaMPI Workshop, SC26: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2026. The final published version will be available in IEEE Xplore.

grained concurrency that task-based models aim to exploit, and precisely the drawback that zero-cost synchronization addresses by incorporating algorithmic knowledge of global communication patterns [1].

In this paper, we bridge the structured semantics of BSP with the flexibility of task-based data-flow execution through three key contributions:

- Task-Aware LPF (TALPF): a reimplementation of LPF designed to be task-aware. TALPF integrates with task-based runtimes such as OpenMP and OmpSs-2 [6, 7], that preserves the analyzable structure of BSP while enabling overlap of computation, communication, and synchronization when dependencies permit.
- MPI notification-based one-sided communication: We introduce an API that enables applications to wait on notification identifiers while providing explicit control over synchronization granularity. Our design enables both BSP-style coarse-grained synchronization and GASPI-style fine-grained coordination through a unified mechanism. The API is integrated into TAMPI, enabling the development of hybrid BSP+tasking and TAGASPI-style applications directly within the MPI ecosystem, while preserving MPI's portability, mature tooling support, and widespread adoption. To achieve this, we extend the API proposed by Sergent et al. [23].
- TAMPI Two-Sided RDMA (2SR): a novel interface that combines sender-driven communication initiation with lightweight receiver-side completion detection implemented exclusively with one-sided operations. Despite its one-sided foundation, 2SR exposes a familiar two-sided programming interface, facilitating straightforward integration into existing MPI applications.

We evaluate our approach on an HPC system. Microbenchmarks show that our implementation of notifications introduce no additional bandwidth or latency overhead, while application-level benchmarks demonstrate that notification-based implementations consistently outperform the built-in one-sided synchronization of MPI and often surpass two-sided communication. Task-aware variants using OmpSs-2 further enhance scalability by reducing synchronization overhead and enabling effective overlap of computation and communication, with TAGASPI still achieving the best overall results.

By integrating BSP-inspired coarse-grained synchronization and GASPI-like fine-grained synchronization into modern task-based runtimes—and embedding these abstractions within the MPI interface—we present a scalable, composable, and performance-portable programming model for many-core HPC systems.

II. Background

This section briefly introduces the programming models and communication technologies relevant to this work, focusing on OmpSs-2, network fabrics, MPI, GASPI, LPF, and task-aware communication libraries.

A. OmpSs-2

OmpSs-2 [6, 7] is a task-based programming model developed at the Barcelona Supercomputing Center (BSC), extending the capabilities of OpenMP tasking model. It enables programmers to express parallelism through task annotations and data dependencies, allowing the runtime system to automatically enforce correct execution order and optimize scheduling. Unlike traditional fork-join models, OmpSs-2 follows a dataflow execution model, enabling dynamic and fine-grained parallelism without explicit parallel regions.

To implement its task-based model, OmpSs-2 relies on the LLVM compiler infrastructure [4] to process task annotations and translate them into calls to the NODES runtime [5]. This runtime is responsible for managing high-level task features such as task dependencies. While, beneath it, the nOS-V [2] runtime handles hardware resource allocation and task dispatch, ensuring efficient execution across heterogeneous architectures. This layered design separates high-level task management from low-level resource control, achieving both programmability and high performance.

B. Network Fabrics

Modern HPC systems rely on advanced network fabrics to deliver low-latency, high-bandwidth communication across distributed nodes. Interconnects such as InfiniBand provide hardware-level support for Remote Direct Memory Access (RDMA), allowing direct data transfers between nodes without CPU intervention or operating system involvement. This capability is crucial for scaling parallel applications efficiently across large clusters.

InfiniBand Verbs (IBVerbs) [18] is a low-level API that exposes RDMA functionality over InfiniBand. It allows applications to register memory regions, configure queue pairs, and submit asynchronous communication requests with precise control over communication semantics. However, while IBVerbs offers optimal performance, its complexity and hardware-specific details make it impractical for most large-scale applications.

To address these complexities, higher-level frameworks such as libfabric (OFI) [13] abstract over multiple fabric technologies (e.g., InfiniBand, Ethernet, Cray networks) through a unified API. OFI maintains support for advanced capabilities such as RDMA and atomic operations while improving portability and developer productivity.

C. Message Passing Interface (MPI)

Message Passing Interface (MPI) [17] is a widely adopted standard for message-passing communication in HPC. As a specification rather than a concrete implementation, MPI defines a portable and scalable interface, with several mature implementations available. For the development and evaluation of the contributions presented in this paper, we have extended the MPICH implementation [16], known for its performance and robustness.

MPI supports a broad range of communication patterns through its collective, two-sided, and one-sided primitives.

While these primitives provide the flexibility to express various parallel communication schemes, MPI's one-sided communication model presents certain limitations. Specifically, it offers only coarse-grained synchronization mechanisms, or requires the use of *MPI_Groups* to establish more fine-grained control. This added complexity can hinder programmability and limit the practical use of one-sided communication, especially in dynamic or fine-grained parallel applications.

D. GASPI

The GASPI standard [24] provides a high-performance, one-sided communication model, based on InfiniBand Verbs, that enables direct read and write access to the memory of remote ranks via memory segments. GASPI offers the *gaspi_write_notify* primitive, which performs a remote write followed by an immediate notification to the target rank. This notification is delivered as soon as the data has been committed to the remote memory, enabling fine-grained synchronization between ranks. To handle these events, GASPI provides dedicated primitives to wait for and process remote notifications, allowing precise tracking of remote completions without requiring a global synchronization.

All one-sided operations in GASPI are submitted to explicitly managed communication queues, which serve to multiplex communication and manage concurrency. When operations targeting the same rank are issued through the same queue, GASPI guarantees their arrival order, ensuring consistent remote behavior. Additionally, GASPI includes the coarse-grained *gaspi_wait* primitive which blocks until all RMA operations submitted to a given queue are locally complete, signaling that local buffers involved in communication can be safely reused.

E. Lightweight Parallel Foundations (LPF)

The Lightweight Parallel Foundations (LPF) library [25] provides a compact, interoperable communication layer that strictly adheres to the Bulk Synchronous Parallel (BSP) model [26]. The design of LPF follows the principles of model compliance and interoperability, ensuring that algorithms proven optimal under the BSP model remain valid across diverse architectures and programming environments.

LPF defines a minimal set of twelve primitives that provide well-defined asymptotic performance guarantees, enabling predictable and analyzable program behavior. LPF relies on MPI for the initial setup phase, including process discovery and global configuration, while data movement is implemented using RDMA with IBVerbs for high efficiency. Global synchronization and communication progression are explicitly controlled through the *lpf_sync* primitive, ensuring precise coordination of computation and communication phases.

F. Task-Aware libraries

Task-Aware libraries [8] enable the seamless integration of communication and other blocking operations within task-based programming models. Their core objective is to allow operations such as MPI or GASPI communication and synchronization to be embedded within tasks without sacrificing correctness or concurrency. When a task performs a blocking operation, Task-Aware libraries suspend its execution, freeing the underlying CPU to execute other ready tasks. A dedicated polling task is responsible for tracking the operation's completion and resuming or finalizing the task once it is safe to do so.

TAMPI (Task-Aware MPI) [21, 22] focuses on two-sided MPI operations. It provides both blocking and non-blocking task-aware APIs, relying on a polling mechanism to manage MPI request progression while avoiding internal MPI thread-safety issues. Communication operations are not issued directly by tasks but are delegated to the polling task to ensure proper serialization. TAMPI integrates closely with task runtimes like OpenMP and OmpSs-2, supporting dependency tracking and overlap of communication with computation.

TAGASPI (Task-Aware GASPI) [19], in contrast, targets one-sided communication. It builds on GASPI's native support for lightweight notifications, enabling efficient synchronization without global barriers. TAGASPI allows tasks to issue communication directly, while the polling mechanism monitors both local completions and remote notifications to release task dependencies. Its design supports asynchronous, fine-grained communication patterns, such as producer-consumer communication patterns, with minimal runtime overhead.

Together, TAMPI and TAGASPI demonstrate the potential of task-aware communication models to unify high-performance communication with task-based parallelism. By exposing both two-sided and one-sided semantics within a task-centric framework, these libraries enable scalable, portable, and efficient hybrid programming across modern HPC systems.

To facilitate integration with different task-based runtimes, such as OpenMP and OmpSs-2, both TAMPI and TAGASPI rely on the Asynchronous Low-Level Programming Interface (ALPI) [20], which standardizes interactions between the task runtime and task-aware libraries.

III. Task-Aware LPF (TALPF)

BSP-based hybrid applications commonly follow a fork-join model, where computation is parallelized but communication and synchronization are executed sequentially, creating performance bottlenecks. First, the synchronization between computation and communication often delays communication that could otherwise begin earlier. Second, serializing the communication phase underutilizes available resources, especially when parallel communication is possible.

Prior research [19, 21, 22] has demonstrated that expressing communication as tasks can significantly improve performance by enabling overlap with computation. Building on this insight, we extend the taskification paradigm to the Lightweight Parallel Foundations (LPF) model, allowing both communication and synchronization to be executed asynchronously and to overlap. As a result, our approach achieves full overlap across

all three phases of the BSP model, further enhancing efficiency and scalability.

To further reduce overhead, we introduce relaxed synchronization modes that eliminate explicit barriers while ensuring correctness. Unlike typical task-aware libraries, TALPF is not a wrapper but a full reimplementation of LPF, enabling deeper runtime integration while maintaining a largely unchanged API, in which communication and synchronization operations are renamed with the prefix *talpf*. It supports parallel execution, asynchronous blocking operations, and non-blocking completion checks.

A central change concerns the point at which communication is issued. In the current LPF implementation, communication primitives do not execute operations immediately, instead, they are queued and later processed in the synchronization call at the end of each superstep. TALPF, in contrast, issues communication operations directly at the invocation of the respective primitives, rather than deferring them to the synchronization phase. This redesign facilitates overlap between communication and computation similarly to the hpget and hpput primitives in classic BSPlib [14], requires more rigorous management of remote memory access to ensure correctness and prevent conflicts, and loses the arbitrary-order write conflict resolution of the original LPF library.

Communication primitives in TALPF are non-blocking. After issuing an operation, a task holds its dependencies and delegates completion tracking to a polling task, which releases them once the operation finishes. This ensures that dependent tasks are only executed once the communication is complete, preserving correctness while maximizing concurrency.

The synchronization primitive in TALPF has been redesigned to support non-blocking behavior. The original LPF implementation involves multiple phases: metadata exchange, write conflict resolution, a second metadata exchange, data transfer, local completion wait, and a final barrier. In TALPF, since communications are already issued, the write-conflict resolution step is removed from the synchronization primitive. Instead, two new explicit atomic operations—`fetch & add` and `compare & swap`—are introduced. Synchronization tasks depend on all preceding communication tasks, ensuring that local operations are complete without requiring an explicit wait.

The new synchronization phase (Table I) thus involves only a message count exchange, remote completion checks, and a final barrier, all handled asynchronously by the polling task. Like communication tasks, synchronization tasks leverage this polling mechanism to maintain asynchronicity and avoid resource wastage due to busy waiting.

The polling task is a cornerstone of TALPF. Executed periodically, it fulfills three main roles: (1) monitoring local completions and releasing communication task dependencies, (2) checking remote completions, and (3) managing asynchronous synchronization steps such as metadata exchange and barriers. This design ensures efficient and fully asynchronous execution across all phases of hybrid BSP applications.

This taskified and asynchronous design makes TALPF a suitable communication layer for dynamic and fine-grained hybrid applications, enabling performance improvements over traditional BSP models while retaining model-compliance and portability.

A. Zero-Cost Synchronization modes

Global synchronization in distributed systems is inherently expensive. To further reduce overhead in TALPF, we introduce zero-cost synchronization modes (Table I)—lightweight alternatives that eliminate all-to-all operations such as message count broadcasts and barriers. These modes restrict synchronization to only directly communicating processes and leverage either application repetition or user-provided information.

In *cached mode*, the first iteration records the expected number of messages, which is reused in subsequent iterations. This avoids repeated collective operations and barriers, enabling true zero-cost synchronization. It is ideal for iterative applications with regular communication patterns.

In *msg mode*, the programmer manually specifies expected message counts, allowing asynchronous execution without collective operations or global synchronization. While more flexible, it shifts responsibility to the user, trading programmability for performance.

Finally, *barrier mode* is not a synchronization mode in itself but an auxiliary mechanism that can be combined with either *cached* or *msg mode* to enforce a global asynchronous barrier when full-system synchronization is required.

Together, these modes offer a configurable balance between performance and ease of use, allowing TALPF to adapt to both repetitive and dynamic communication patterns efficiently.

IV. Enhance MPI & TAMPI One-sided

This section presents our proposed API, which provides notification-based waiting mechanisms with explicit control over synchronization granularity. Our approach supports both coarse- and fine-grained synchronization, enabling applications to adapt to diverse communication patterns while eliminating the need for global barriers and improving overall concurrency. This flexibility allows programmers to express a wide spectrum of coordination strategies within a single, coherent model. The proposed design builds upon and extends the API introduced by Sergent et al. [23].

This receiver-driven coordination naturally enables fine-grained synchronization and is particularly well suited to irregular communication patterns. Moreover, on modern network interconnects, it can be implemented with no additional overhead compared to the original `MPI_Put` operation.

To integrate this mechanism into task-based runtimes, we extend the Task-Aware MPI (TAMPI) library. Leveraging TAMPI's external events and polling infrastructure, tasks can depend on the completion of one-sided operations and notifications without blocking, preserving dependency semantics and maximizing parallelism. The extension maintains TAMPI's philosophy of non-intrusive task integration, ensuring that communication progress and dependency management remain efficient and transparent.

TABLE I: Synchronization modes of LPF and TALPF

Synchronization Mode	Meta-Data / Control Exch.	Write Conflict Resolution	Compute #Messages	Comm. Data Exchange	Wait Local Completion	Wait Remote Completion	Barrier
DEFAULT (LPF)	Synchronous	Yes	No	Yes	Yes	No	Sync.
DEFAULT (TALPF)	Simplified & Asynchronous	No	Collective Exchange	No	No	Yes	Async.
CACHED (TALPF)	No	No	Cached Value	No	Yes	No	No
MSG (TALPF)	No	No	Parameter Value	No	Yes	No	No
BARRIER (TALPF)	-	-	-	-	-	-	Async.

We also introduce TAMPI Two-Sided RDMA (2SR), a task-aware API built on top of TAMPI that mimics traditional two-sided semantics using one-sided operations. The enabling insight here is that by providing global knowledge about the communication pattern, local processes can determine completion of communication phases without any additional global communication. This idea was previously coined under the name *zero-cost synchronization* [1]. Our implementation of this idea, 2SR, combines explicit notifications with TAMPI's dependency tracking to provide safe efficient synchronization while retaining the simplicity of two-sided communication and the performance of RDMA.

A. Notification API extension

To achieve the desired synchronization mechanism, the current MPI one-sided API is insufficient. Specifically, enabling lightweight and explicit synchronization for one-sided communication requires the ability to attach a tag to communication operations, allowing the origin rank to notify the target rank upon completion. Additionally, dedicated operations are needed for the receiver to check or wait for these notifications. A key design goal of our desired API is to remain lightweight and simple, avoiding the need to initialize additional objects or invoke extra operations beyond the communication and the synchronization itself.

Sergent et al. [23] proposed a GASPI-inspired API extension to MPI one-sided communication that satisfies our requirements. Figure 1 illustrates the proposed API, with our extensions highlighted in bold. The original API augments `MPI_Put` and `MPI_Get` with a *notification_id* (tag) parameter—`MPIX_Get_notify` is omitted from the figure for simplicity—and introduces two synchronization primitives, `MPIX_Win_wait_notify` and `MPIX_Win_test_notify`, which waits or checks for the arrival of a given *notification_id*.

Our extension incorporates a counter parameter into the synchronization calls, enabling applications to wait for multiple occurrences of the same notification and thereby explicitly control synchronization granularity. In addition, we introduce `MPIX_Notify` as a lightweight mechanism for issuing notifications; semantically, it behaves as an empty put operation. While this fine-grained control can be emulated with the original API through multiple `MPIX_Win_{wait, test}_notify` calls, consolidating it into a single call improves not only programmability but also performance, particularly in the Task-Aware domain.

```
int MPIX_Put_notify(const void *origin_addr,
        int origin_count,
        MPI_Datatype origin_datatype,
        int target_rank,
        MPI_Aint target_disp,
        int target_count,
        MPI_Datatype target_datatype,
        MPI_Win win, int tag);

int MPIX_Notify(int target_rank,
        MPI_Win win, int tag);

int MPIX_Win_wait_notify(MPI_Win win,
        int tag, int count);

int MPIX_Win_test_notify(MPI_Win win,
        int tag, int count, int *flag);
```

Fig. 1: MPI one-sided API proposed by Sergent et al., with our extensions highlighted in bold.

Both wait and test functions may be extended to support waiting on or testing for multiple tags simultaneously, following a design similar to how arrays of MPI_Request objects are managed in the standard MPI API.

B. Implementation

We implemented the proposed API using MPICH as the base MPI implementation, specifically leveraging the CH4 OFI netmod. The OFI provider, which is based on Verbs, satisfies all the requirements imposed by both MPICH and our API design. The OFI layer enables the attachment of a 64-bit unsigned integer (or 32-bit when using certain Verbs-based backends) directly to a communication operation. This feature is convenient, because it allows the notification tag to be embedded within the communication itself, thereby eliminating the need for auxiliary metadata structures, effectively providing a *zero-cost synchronization* mechanism [1]. The tag is transmitted through the network alongside the operation and becomes available upon its completion.

However, if the underlying backend or provider does not support this mechanism, our implementation approach cannot be applied. In such cases, alternative methods must be employed. For example, Sergent et al. [23] propose using two linked messages to achieve comparable functionality.

Upon completion of the communication, an entry is generated in the receiver's completion queue. The target rank polls this queue to retrieve completions, from which it extracts the notification tag. The tag is then used to update an internal notification hash table: either a new entry is created or the value of an existing entry is incremented. The `MPIX_Win_test_notify` and `MPIX_Win_wait_notify` operations query this map to determine whether the required number of notification occurrences has been received and, if so, decrement the hash table value accordingly.

C. TAMPI One-Sided Extension

We integrated the proposed notification-based API into TAMPI to provide task-awareness and seamless interoperability with task-based programming models. Two variants of the API were implemented: a blocking version, in which communication operations pause the executing task until completion, and a non-blocking version, which instead defers the release of task dependencies while allowing the task itself to continue execution. This enables efficient overlap of communication and computation while preserving data-flow correctness.

D. TAMPI Two-Sided RDMA (2SR)

Using the one-sided API in real-world applications is not always straightforward, particularly when adapting an existing two-sided implementation. A key challenge arises from the risk of unexpected memory overwrites, where data may be written before the receiving side is ready to handle it. This typically necessitates an acknowledgment mechanism to indicate when the source rank is allowed to perform the write. To address this issue, we designed and implemented the TAMPI Two-Sided RDMA (2SR) API (Figure 2), a lightweight abstraction that mimics the semantics and the programmability of `MPI_Send/MPI_Recv` calls using exclusively one-sided operations, while preventing data races. Note that, in order to maintain the `MPI_Recv` semantics, `TAMPI_2SR_Recv` includes in its parameters a buffer, the number of elements to receive (count) and the elements datatype, even if the parameters are unused.

Figure 3, shows a comparison between MPI two-sided, MPI one-sided with notifications, and MPI 2SR, for simplicity, the diagrams do not show task-awareness. The `TAMPI_2SR_Send` function performs a remote memory write only after synchronize with the matching `TAMPI_2SR_Recv`. Specifically, `TAMPI_2SR_Send` begins by waiting for a notification from the corresponding `TAMPI_2SR_Recv`. Once the notification is received, it proceeds to transfer the data with a notification to the receiver. On the receiving side, `TAMPI_2SR_Recv` first sends an initial notification to indicate readiness, and then waits for the notification signaling that the data has arrived. In contrast to the eager approach of MPI two-sided, we require the receiver to start the communication as the put operation writes the message directly to the final destination.

```
int MPI_Send(const void *buf, int count,
        MPI_Datatype datatype, int dest,
        int tag, MPI_Comm comm);

int TAMPI_2SR_Send(void *buf, int count,
        MPI_Datatype datatype, int dest,
        int disp, int tag, MPI_Win win);

int MPI_Recv(void *buf, int count,
        MPI_Datatype datatype, int src,
        int tag, MPI_Comm comm,
        MPI_Status * status);

int TAMPI_2SR_Recv(void *buf, int count,
        MPI_Datatype datatype, int src,
        int tag, MPI_Win win);
```

Fig. 2: Comparison of MPI Two-sided API and TAMPI 2SR API.

This approach effectively protects remote memory from unsolicited writes, as communication can only proceed after explicit synchronization initiated by the receiver. However, while it restores the intuitive safety of two-sided semantics, this approach trades off some of the flexibility and performance benefits typically associated with one-sided communication.

V. Evaluation

In this section, we present an extensive evaluation of our proposed implementation on a high-performance computing (HPC) system. We begin by detailing the hardware and software specifications of the benchmarks used in our experiments. Then we evaluate the bandwidth and latency of our notification-based MPI extensions, and we compare them against the current MPI implementations. Finally, we introduce a set of benchmarks designed to assess the performance of our implementations and compare them against the current implementations from MPI, TAMPI and TAGASPI.

A. Methodology

To evaluate the performance of our proposed implementation, we conducted experiments on an HPC system. Each compute node in this system is equipped with two Intel Sapphire Rapids 8480+ processors, offering a total of 112 physical cores per node (56 cores per CPU). The nodes are interconnected via a NDR200 (shared between two nodes), which implements a 200 Gb/s InfiniBand network (100Gb/s per node), enabling high-throughput, low-latency communication across the system. This architecture provides an ideal environment to assess the scalability and efficiency of our notification-based MPI extensions under realistic HPC workloads.

All results are obtained from executions on an isolated machine avoiding external sources of variability, and each experiment was repeated multiple times to ensure the correctness of our results.

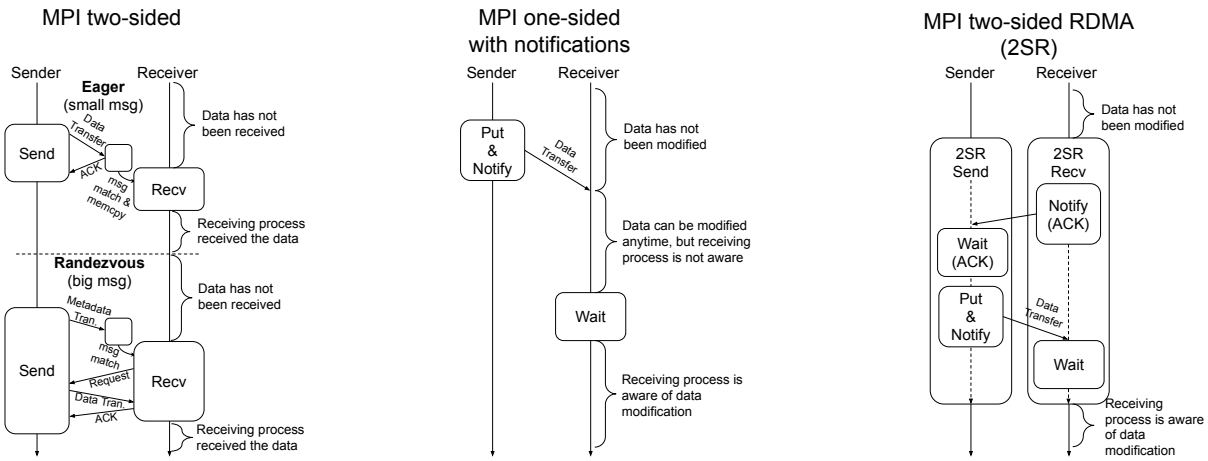


Fig. 3: Diagram illustrating the communication process, memory states and the message awareness of the receiving process using MPI two-sided (left), MPI one-sided with notifications (center), and MPI two-sided RDMA (right).

Pure SPMD versions run with one MPI rank per core (112 ranks per node), while Task-Aware (hybrid) versions run with one MPI rank per socket (2 ranks per node) and rely on OmpSs-2 to manage and schedule threads within each rank; In both cases, ranks and threads are mapped to the available cores to avoid oversubscription.

B. MPI Micro-benchmarks Evaluation

Before evaluating our implementation using full benchmarks, we conducted a preliminary analysis of bandwidth and latency. This initial step provides a baseline for understanding the communication performance and helps isolate the impact of synchronization and notification mechanisms in subsequent experiments.

We implemented two test versions, each comprising both a baseline and our approach. The first version corresponds to ping-pong tests, where we compare a traditional two-sided MPI approach with our one-sided MPI approach using notifications. In the two-sided version, the first process sends data and then receives it back, while the second process first receives the data and subsequently sends it back. In contrast, the one-sided version employs notification-based operations: the first process performs a put with a notification and then waits to receive a notification in return; the second process waits for this notification before responding with its own put with a notification. The second test set measures raw data transfer performance by performing multiple consecutive put operations without explicit synchronization, timing only the data transfer. For this test, we compare executions using the standard `MPI_Put` operation against those using the new `MPI_Put_notify` operation.

Figure 4 presents the bandwidth results (in GB/s). Three main observations can be made:

1) Adding a notification to `MPIX_Put` does not affect performance, as the curves for MPI RMA Put (no sync) and MPI RMA Put Notify (no sync) overlap across all message sizes.
2) The two-sided model (MPI MSG (ping-pong)) has a similar performance to the one-sided version (MPI RMA Notifications (ping-pong)).
3) The cost of one-sided synchronization becomes evident when comparing the bandwidth of RMA operations without synchronization against the ping-pong version, which includes synchronization.

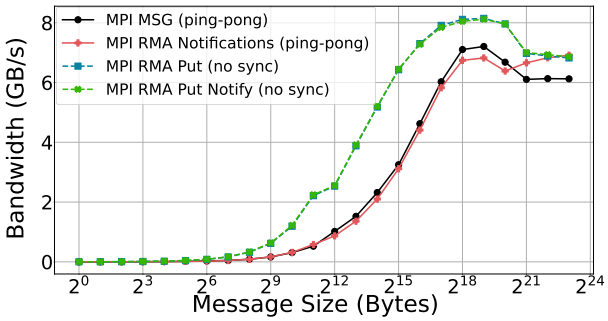


Fig. 4: Bandwidth comparison between MPI two-sided (MPI MSG) and MPI one-sided with notifications (MPI RMA Notifications) in a ping-pong benchmark, and between MPI one-sided Put (MPI RMA Put) and MPI one-sided Put with notification (MPI RMA Put Notify) without synchronization. Higher is better.

Figure 5 reports the latency results (in µs), and the same trends emerge. First, adding a notification in `MPIX_Put` does not introduce additional latency. Second, MPI two-sided operations remain slightly more efficient than their one-sided counterparts in the ping-pong test. Third, the overhead of synchronization is clearly observable.

Additionally, the figure highlights that latency for pure Put operations remains stable up to a message size of 2 KB; beyond this point, it increases linearly with message size. In the ping-pong versions, this growth is smoother around the transition but still follows the same trend.

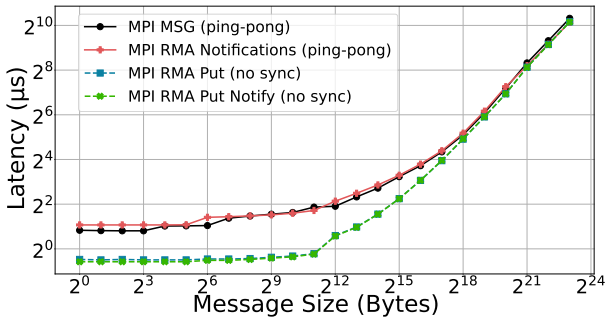


Fig. 5: Latency comparison between MPI two-sided (MPI MSG) and MPI one-sided with notifications (MPI RMA Notifications) in a ping-pong benchmark, and between MPI one-sided Put (MPI RMA Put) and MPI one-sided Put with notification (MPI RMA Put Notify) without synchronization. Lower is better.

C. Benchmarks Evaluation

To evaluate the performance and flexibility of our proposed extensions, we selected four representative benchmarks: Jacobi heat diffusion, N-Body, Matrix Multiplication, and a Streaming application. These workloads cover a diverse set of communication patterns and granularities. Jacobi and Matrix Multiplication use row-based data decomposition, resulting in a high volume of messages per iteration. Jacobi adheres to a traditional BSP-style pattern, whereas Matrix Multiplication and N-Body follow a ring-based communication scheme. In contrast, Streaming does not have any computation, and employs element-wise decomposition and a two-phase communication model that resembles a data-flow pattern with acknowledgment. Collectively, these benchmarks enable a comprehensive evaluation of our system under both coarse- and fine-grained synchronization scenarios, as well as structured and dynamic communication patterns.

Over these benchmarks, we implemented the following versions:

- **(TA)MPI MSG**: A baseline MPI implementation using two-sided communication via *MPI_Send* and *MPI_Recv* primitives. Pure SPMD and Task-Aware versions.
- **MPI RMA Fence**: An MPI one-sided implementation using `MPI_Put` and `MPI_Win_fence` for global synchronization at each iteration. Pure SPMD version.
- **MPI RMA PSCW**: An MPI one-sided implementation using `MPI_Put` with the post-start-complete-wait (PSCW) synchronization model: `MPI_Win_{post, start, complete, wait}`. Pure SPMD version.
- **(TA)MPI RMA Noti**: Our MPI RMA implementation using `MPI_Put_notify` for communication, and a single call to `MPI_Win_wait_notify` to wait for all notifications. Pure SPMD and Task-Aware versions.
- **(TA)MPI RMA Noti Fine**: Similar to MPI RMA Noti, but with fine-grained synchronization where each notification is awaited individually via separate calls to `MPI_Win_wait_notify`. Pure SPMD and Task-Aware versions.
- **TAMPI RMA 2SR**: An implementation using the proposed Two-Sided RMA (2SR) primitives, built on top of TAMPI to emulate two-sided semantics using one-sided operations with notifications. Task-Aware version.
- **TALPF**: A baseline implementation using the TALPF library with its default global synchronization mode. Task-Aware version.
- **TALPF Cached**: A TALPF implementation using the cached synchronization mode, which reuses synchronization metadata across iterations to reduce overhead. Task-Aware version.
- **TAGASPI**: An implementation using the TAGASPI library, based on one-sided communication with native notification support for fine-grained synchronization. Task-Aware version.

All the Task-Aware versions use non-blocking APIs whenever possible. Both the pure SPMD and Task-Aware versions were executed on 16 nodes and 2 nodes, comprising a total of 1,792 and 224 cores, respectively. However, LPF was unable to complete the pure SPMD runs on 16 nodes, due to an out-of-memory error caused by excessive metadata memory allocations.

We scaled the problem sizes proportionally to the number of cores in each configuration. For Jacobi, we used 56k and 7k square matrices for 16 and 2 nodes, respectively, with 512 iterations. In N-body, we used 3,584k and 448k particles, running 16 iterations. In Matmul, we again used 56k and 7k square matrices. Finally, in Streaming, we used 448k and 56k elements for 8,136 iterations. We evaluated multiple block sizes and show the best-performing ones in Figures 6 and 7.

Figure 6 shows the normalized performance of each version across our benchmark suite using 16 nodes. Here, normalized performance is defined as the performance of a given version divided by the performance of the best-performing version for that benchmark. Error bars indicate the standard deviation.

Figure 7 shows the normalized performance of each version across the benchmark suite using 2 nodes. Overall, we observe trends similar to those in Figure 6, with reduced performance. In contrast to the 16-node case, results for the pure SPMD version of LPF are also available at this scale.

Our MPI Notification implementations—both fine-grained and coarse-grained synchronization—generally outperform MPI's built-in RMA synchronization methods. With the exception of the streaming benchmark, they also surpass the MPI two-sided implementation (MPI MSG). In the streaming benchmark, however, the coarse-grained approaches exhibit performance degradation due to the additional delay introduced in the first iteration. Specifically, in the fine-grained versions (MPI MSG and MPI RMA Noti fine), each rank only waits for the arrival of its first message before proceeding. In contrast, the coarse-grained approaches require all messages to be received before any rank can advance, creating a cascade effect that amplifies delays during the first iteration.

Pure SPMD approaches come with inherent trade-offs: they enable fully parallel communication but suffer from tighter

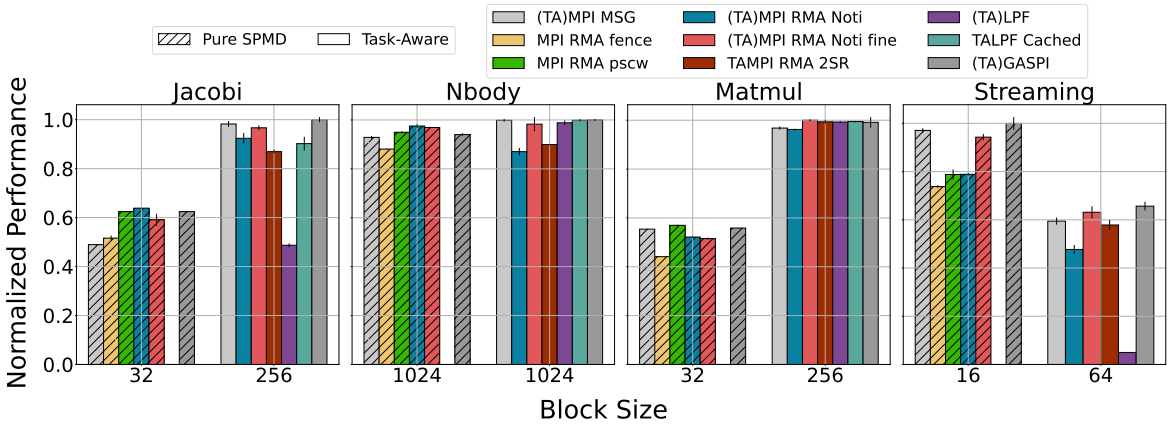


Fig. 6: Normalized performance evaluation across various benchmarks with optimal granularities, using 16 nodes (1792 ranks with pure SPMD versions and 32 ranks with Task-Aware versions). Higher is better.

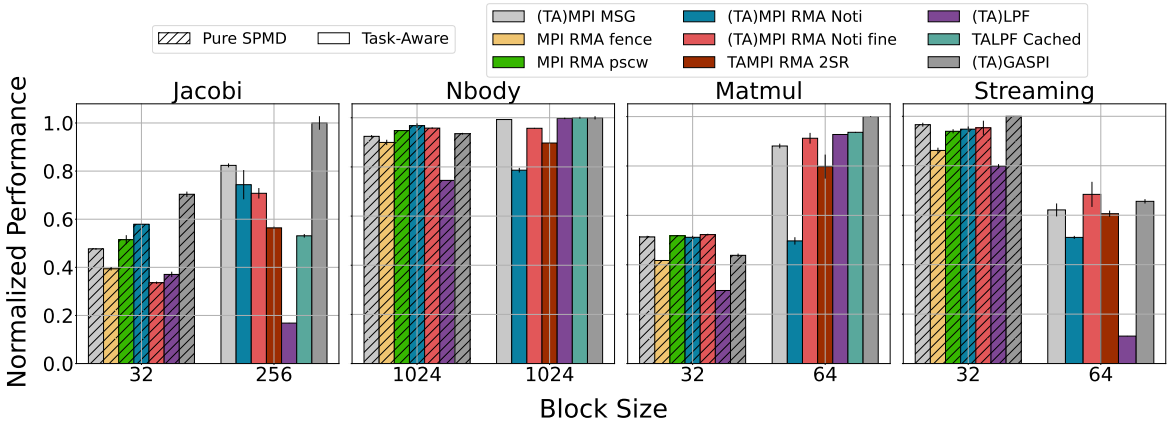


Fig. 7: Normalized performance evaluation across various benchmarks with optimal granularities, using 2 nodes (224 ranks with pure SPMD versions and 4 ranks with Task-Aware versions). Higher is better.

synchronization requirements and limited flexibility (communication must be issued in strict order with no overlap between computation and communication). Additionally, when extra memory is needed, it must be replicated across all ranks.

By contrast, Task-Aware implementations mitigate several of these drawbacks. They reduce synchronization and memory overheads by involving fewer ranks, allow more flexibility by issuing communication as soon as tasks are ready, and enable effective overlap of computation, communication, and synchronization. Computation remains fully parallel, and communication calls to the runtime can also be issued in parallel. The main limitation is that, in TAMPI, communication operations that were originally distributed across many ranks are now serialized onto fewer ranks (e.g. 56 ranks of communication executed in parallel in the pure SPMD version are executed by a single rank in the Task-Aware version). This drawback is especially visible in the streaming benchmark where there is practically no computation, and in the coarse-grained one-sided Matmul, where serialized transfers of submatrices and the absence of fine-grained synchronization substantially degrade performance when working with small matrices.

Among the Task-Aware variants, TAGASPI, due to a more specialized implementation, consistently achieves the highest performance, followed closely by TALPF (using cached synchronization), then TAMPI with two-sided communication, and finally TAMPI's one-sided notifications in fine-grained mode. Overall, Task-Aware (hybrid) implementations generally outperform their flat MPI counterparts. Across both flat and Task-Aware versions, fine-grained synchronization consistently outperforms coarse-grained synchronization.

Among the Task-Aware variants, TAGASPI consistently delivers the highest performance, owing to its specialized implementation. TALPF with cached synchronization follows closely, while TAMPI with one-sided notification-based implementation in fine-grained mode achieves performance comparable to TAMPI's two-sided communication. Overall, Task-Aware (hybrid) implementations consistently outperform their flat MPI counterparts, demonstrating the benefits of overlapping communication and computation. Furthermore, across both flat and Task-Aware implementations, fine-grained synchronization consistently achieves better performance than coarse-grained synchronization.

VI. Related Work

The lack of efficient fine-grained synchronization in MPI's one-sided communication has long been recognized by the HPC community. This limitation has led to several proposals aimed at extending or improving MPI one-sided semantics.

Belli and Hoefler [9] introduced Notified Access, which augments RMA operations with tags and persistent notification requests to track remote completions. Their approach enables low-overhead, fine-grained synchronization and improves performance over traditional methods.

Intel MPI introduced its own proprietary notification-based extension to MPI one-sided [15], providing developers with a mechanism to explicitly signal and detect the completion of remote operations, demonstrating the practical demand for such functionality in production environments.

An open issue within the MPI Forum [12], highlights the absence of native notification support in the standard. Efficient synchronization patterns, such as producer-consumer models, require a mechanism that enables the target process to detect the completion of remote memory operations reliably and with low overhead. This issue has recently been updated with a proposal, Notified RMA, which follows an API similar to Intel MPI. The main difference with our approach lies in the handling of notifications on the receiver side. The proposal introduces a get/set mechanism for the *notification_id* counter within a window, which is incremented for each one-sided communication operation associated with that identifier.

While Notified RMA is still under active discussion, their proposal and ours are not mutually exclusive and can naturally coexist within the same standard. In particular, Notified RMA targets a lower-level interface, whereas our API provides a higher-level abstraction for notification handling, offering a more structured approach to synchronization. However, the current Notified RMA design may be susceptible to race conditions, as a notification arriving between the get and set operations could be overwritten by the set, leading to lost updates. This could be mitigated by using atomic or decrement-based operations to ensure greater robustness.

Finally, Sergent et al. [23] proposed a GASPI-inspired notification mechanism that introduces fine-grained, receiver-driven synchronization operations, enabling more expressive and efficient coordination of communication events.

On the side of BSP compliant programming models, including BSPlib [14] and LPF [25], a matching concept that reduces synchronisation cost in the same way we do is given by Alpert and Philbin [1]–as far as we know, we are the first implementation of their zero-cost synchronization concept.

Our work extends these ideas with a portable MPI API that supports fine- and coarse-grained notification-based synchronization while integrating seamlessly with task-based runtimes via TAMPI. A key feature of our approach is the support for waiting on multiple occurrences of the same notification, providing fine control over the granularity of synchronization and enabling more efficient coordination in iterative or streaming communication patterns. This task-aware integration allows communication and synchronization to proceed without blocking, making it well-suited for modern hybrid applications.

VII. Conclusions

In this work, we address key limitations in hybrid parallel programming by introducing mechanisms that improve the integration of one-sided communication and synchronization within task-based runtimes. We introduce TALPF, a relaxed BSP model that exposes overlap between computation, communication, and synchronization, together with a lightweight notification-based MPI extension. Integrated into TAMPI and complemented by the 2SR abstraction, these mechanisms enable task-aware, portable, and low-overhead synchronization by leveraging runtime knowledge to reduce coordination costs.

Our evaluation reveals a clear specialization of models: two-sided MPI excels in latency-bound or communication-only workloads, coarse-grained one-sided communication fits regular BSP patterns, and fine-grained notifications are best suited for irregular and dynamic applications. LPF offers strong predictability but limited scalability under strict synchronization, whereas TALPF removes this limitation by relaxing phase boundaries and enabling runtime-driven overlap. This leads to significant gains when computation is available to hide communication, while in communication-dominated or fine-grained workloads, tasking overheads and limited overlap opportunities may reduce performance relative to SPMD.

Overall, TALPF bridges BSP and task-based execution, preserving structure while unlocking concurrency and scalability through asynchronous progress and zero-cost synchronization modes. The proposed notification mechanism further improves flexibility for fine-grained coordination with negligible overhead, and 2SR provides efficient two-sided semantics on top of one-sided operations. Across all experiments, task-aware variants consistently outperform SPMD executions in compute-communication overlap scenarios, while TAGASPI remains the top-performing baseline, closely matched by fine-grained task-aware models without global synchronization.

These results demonstrate that combining task-awareness with one-sided communication is a practical and portable path toward scalable HPC execution. More broadly, they establish that a unified synchronization layer spanning BSP semantics, fine-grained coordination, and task-based runtimes is key to achieving scalable, composable, and high-performance hybrid programming on modern many-core systems.

Acknowledgment

This work was partially supported by the Generalitat de Catalunya (contract 2021-SGR-01007) and the Spanish Government through the Severo Ochoa Program (CEX2021-001148-S/MCIN/AEI/10.13039/501100011033). This work was also supported by the ST4HPC research project (PID2023-147979NB-C21), funded by MICIU/AEI/10.13039/501100011033 and by FEDER, UE, and in part by the HPC Technology Innovation Lab, a research cooperation agreement between the Barcelona Supercomputing Center and Huawei (2020–2023).

References

[1] Richard Alpert and James Philbin. "cBSP: Zero-cost synchronization in a modified BSP model". In: *NEC Research Institute, Princeton, NJ, USA, Tech. Rep* (1997), pp. 97–054.

[2] David Álvarez, Kevin Sala, and Vicenç Beltran. "nOS-V: Co-Executing HPC Applications Using System-Wide Task Scheduling". In: *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2024, pp. 312–324. DOI: 10.1109/IPDPS57955.2024.00035.

[3] Pavan Balaji et al. "MPI on millions of cores". In: *Parallel Processing Letters* 21 (2011), pp. 45–60. DOI: 10.1142/S0129626411000060.

[4] Barcelona Supercomputing Center, STAR group. *LLVM-based compiler for OmpSs-2*. Accessed: 2026-05-07. URL: https://github.com/bsc-pm/llvm.

[5] Barcelona Supercomputing Center, STAR group. *NODES: nOS-V Based OmpSs-2 Dependency System*. Accessed: 2026-05-07. URL: https://github.com/bsc-pm/nodes.

[6] Barcelona Supercomputing Center, STAR group. *OmpSs-2 Releases*. Accessed: 2026-05-07. URL: https://github.com/bsc-pm/ompss-2-releases.

[7] Barcelona Supercomputing Center, STAR group. *Specification of OmpSs-2 2025.11*. Accessed: 2026-05-07. URL: https://pm.bsc.es/ompss-2-docs/spec.

[8] Barcelona Supercomputing Center, STAR group. *Task-Aware libraries repository*. Accessed: 2026-05-07. URL: https://github.com/bsc-pm/ta-x.

[9] Roberto Belli and Torsten Hoefler. "Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization". In: *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2015, pp. 871–881. DOI: 10.1109/IPDPS.2015.30.

[10] R.H. Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[11] Hadi Esmaeilzadeh et al. "Dark silicon and the end of multicore scaling". In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 365–376.

[12] MPI Forum. *MPI "RMA Notification" GitHub Issue*. https://github.com/mpi-forum/mpi-issues/issues/59. Accessed: 2026-05-07. 2016. URL: https://github.com/mpi-forum/mpi-issues/issues/59.

[13] OpenFabrics Interfaces Working Group. *Libfabric Programmer's Manual*. Accessed: 2026-05-07. URL: https://ofiwg.github.io/libfabric/.

[14] Jonathan M. D. Hill et al. "BSPlib: The BSP programming library". In: *Parallel Comput.* 24.14 (1998). ISSN: 0167-8191. DOI: 10.1016/S0167-8191(98)00093-3.

[15] Intel Corporation. *Notified One-Sided Communications*. Intel MPI Library Developer Guide for Linux. Intel. URL: https://www.intel.com/content/www/us/en/docs/mpi-library/developer-guide-linux/2021-16/notified-one-sided-communications.html.

[16] Argonne National Laboratory. *MPICH: High-Performance Portable MPI*. Accessed: 2026-05-07. URL: https://www.mpich.org/.

[17] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 5.0*. June 2025. URL: https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf.

[18] *OpenFabrics Alliance, "libibverbs - OpenFabrics Verbs API."*. Accessed: 2026-05-07. URL: https://github.com/linux-rdma/rdma-core.

[19] Kevin Sala, Sandra Macià, and Vicenç Beltran. "Combining One-Sided Communications with Task-Based Programming Models". In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 2021, pp. 528–541. DOI: 10.1109/Cluster48925.2021.00024.

[20] Kevin Sala et al. "ALPI: Enhancing Portability and Interoperability of Task-Aware Libraries". In: *Asynchronous Many-Task Systems and Applications*. Ed. by Patrick Diehl et al. Cham: Springer Nature Switzerland, 2024, pp. 142–153. ISBN: 978-3-031-61763-8.

[21] Kevin Sala et al. "Improving the Interoperability between MPI and Task-Based Programming Models". In: *Proceedings of the 25th European MPI Users' Group Meeting*. EuroMPI '18. 2018. DOI: 10.1145/3236367.3236382.

[22] Kevin Sala et al. "Integrating blocking and non-blocking MPI primitives with task-based programming models". In: *Parallel Comput.* 85 (2019), pp. 153–166. DOI: 10.1016/J.PARCO.2018.12.008.

[23] Marc Sergent et al. "Efficient notifications for MPI one-sided applications". In: *Proceedings of the 26th European MPI Users' Group Meeting*. EuroMPI '19. Association for Computing Machinery, 2019. DOI: 10.1145/3343211.3343216.

[24] Christian Simmendinger, Mirko Rahn, and Daniel Gruenewald. "The GASPI API: A Failure Tolerant PGAS API for Asynchronous Dataflow on Heterogeneous Architectures". In: *Sustained Simulation Performance 2014*. Ed. by Michael M. Resch et al. Cham: Springer International Publishing, 2015, pp. 17–32. ISBN: 978-3-319-10626-7.

[25] Wijnand Suijlen and A. N. Yzelman. "Lightweight Parallel Foundations: a model-compliant communication layer". In: *CoRR* abs/1906.03196 (2019). arXiv: 1906.03196. URL: http://arxiv.org/abs/1906.03196.

[26] Leslie G. Valiant. "A bridging model for parallel computation". In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. DOI: 10.1145/79173.79181.