MPI Progress For All

Hui Zhou*, Robert Latham*, Ken Raffenetti*, Yanfei Guo* and Rajeev Thakur*
*Argonne National Laboratory
Lemont, IL 60439, USA

***Abstract*—The progression of communication in the Message Passing Interface (MPI) is not well defined, yet it is critical for application performance, particularly in achieving effective computation and communication overlap. The opaque nature of MPI progress poses significant challenges in advancing MPI within modern high-performance computing practices. First, the lack of clarity hinders the development of explicit guidelines for enhancing computation and communication overlap in applications. Second, it prevents MPI from seamlessly integrating with contemporary programming paradigms, such as task-based runtimes and event-driven programming. Third, it limits the extension of MPI functionality from user space. In this paper, we examine the role of MPI progress by analyzing the implementation details of MPI messaging. We then generalize the asynchronous communication pattern and identify key factors influencing application performance. Based on this analysis, we propose a set of MPI extensions designed to enable users to construct and manage an efficient progress engine explicitly. We compare our approach to previous efforts in the field, highlighting its reduced complexity and increased effectiveness.**

I. Introduction

Overlapping computation and communication [1], [2] is a key performance goal in high-performance computing (HPC). Ideally, with 100% computation/communication overlap, communication and synchronization become effectively free, allowing parallel applications to scale perfectly. However, achieving this overlap goal remains challenging. The Message Passing Interface (MPI), the *de facto* communication runtime for HPC applications, does not precisely define how communication progress is made. MPI guarantees that once communication is initiated, it will complete, but it does not specify whether progress occurs during the starting call (*e.g.,* `MPI_Isend`), the completion call (*e.g.,* `MPI_Wait`), or in between. To achieve effective computation/communication overlap, strong progress from MPI is desirable [3], meaning that MPI can make progress between the starting and completion calls without explicit MPI calls from the user. However, implementing strong progress poses constraints on MPI implementations and is not always feasible.

The obvious approach to strong progress is to seek hardware solutions. For example, remote direct memory access (RDMA) technology allows network hardware to communicate without the operating system or CPU involvement. Hardware solutions, however, have their own limitations. Hardware may have limited capacity to handle unexpected messages and congestion, lack semantic contexts to make smart routing decisions, and lack mechanisms to provide feedback to the software layer. In addition, hardware solutions are always system-dependent. The arrival of GPU architectures presents new challenges. GPU Direct RDMA technology is still in the emerging stage and cannot always be assumed to be available. It also poses high setup cost and latency. Alternate solutions such as GDRCopy or software pipelining may provide better performance but require weak progress.

Thus, a performance-portable application that relies on good computation/communication overlap generally requires an explicit progression scheme to regularly invoke MPI progress during computation.

Applications currently have limited means to explicitly control MPI progress. The conventional method to invoke MPI progress is via `MPI_Test`. However, `MPI_Test` is tied to a specific MPI request. Thus, designing a progress engine that includes MPI progress requires a synchronization mechanism for managing MPI requests, which is often complex and prone to inefficiency. This is particularly problematic with the asynchronous programming paradigm, including task-based [4], [5] and event-driven programming [6]. Asynchronous programming systems typically include a software progress engine that provides the service of task scheduling and event dispatching. Implementing a separate progress engine for MPI breaks the tasks and events abstractions and contends for resources between the progress engines. Unifying the progression requires managing MPI requests. This is not always possible when the task or event-based runtimes are not MPI-aware. Even when it is possible, it adds complexity and is difficult to implement efficiently.

The core issue is the lack of interoperability from MPI progress. First, MPI progress is not explicitly exposed. It is ambiguous on how to invoke MPI progress or whether a regular MPI progress invocation is needed. The semantics of `MPI_Test` is tied to a particular operation represented by an MPI request and only invokes progress as a side effect. To design an effective progress engine, an explicit MPI progress invocation API is needed. Second, there is no effective way to hook into MPI progress so that a progression scheme for MPI can also progress user-defined asynchronous tasks rather than having to create and manage different progression schemes for individual asynchronous tasks. All asynchronous tasks, including MPI operations, share similar patterns. Without mechanisms to hook into MPI progress forces applications to deal with separate progression mechanisms, which not only increases complexity but also affects performance due to contention and necessary synchronization between the progress engines. An interoperable MPI progress would allow an external progress engine, e.g., from a task runtime or an event system, to efficiently progress MPI, or it will allow an

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00063

application to implement a custom MPI progression scheme. Both solutions will also work for user-defined tasks, avoiding duplication and inefficiency due to contention.

Explicit MPI progress is not needed when there is a strong progress guarantee. However, providing an explicit MPI progress interface provides performance portability, since only the implementation knows whether software progress is needed. When there is strong progress, the explicit MPI progress can be reduced to nearly a no-op.

The lack of interoperability from MPI progress is also one of the key barriers keeping MPI from advancing. As HPC enters the exascale era, MPI faces performance challenges on increasingly hybrid node architectures. Achieving high performance with MPI implementations is becoming more complex and challenging compared to hand-tuned, non-portable solutions. Researchers need the ability to prototype MPI algorithms and MPI extensions independently of MPI implementations. The ROMIO project, which prototyped and implemented MPI-IO during MPI-2 standardization [7], is a good example of such an approach. Today, a potential area for similar innovation is MPI collective operations. An algorithm for a collective operation often involves a collection of communication patterns tied together by a progression schedule. An optimized collective algorithm may integrate both MPI communications and asynchronous local offloading steps, tailored to specific system configurations and application needs. Therefore, exposing and making MPI progress interoperable with user-layer asynchronous tasks will stimulate broader community research activities, driving future advancements in MPI. We aim to address a common debate in MPI standardization meetings—whether a proposed feature needs to be *in MPI* or whether it can be a library *on top of MPI*. Exposing new progress APIs will facilitate more tightly coupled libraries, so more features can be built on top of MPI first, rather than directly added into MPI before widespread adoption.

In this work, we introduce a set of MPI extensions to allow applications to explicitly invoke MPI progress without tying to specific communication calls, thereby enabling applications to manage MPI progress without the complexity of handling individual MPI request objects. Our previously introduced MPIX Stream concept [8] is used to target progress to specific contexts, avoiding multithreading contention issues associated with traditional global progress. We also recognize that MPI progress can be extended to collate progress for asynchronous tasks in general, simplifying the management of multiple progress mechanisms and avoiding wasting cycles on maintaining multiple progress engines.

II. Anatomy of Asynchronous Tasks and Role of Progress

Before discussing how to manage progress, we need to define what is progress and understand its role.

A. MPI's Message Modes

To understand the role of progress, we examine how an MPI implementation might send and receive messages. The following discussion is based on MPICH, but we believe it is applicable to other MPI implementations as well.

Figure 1 illustrates various modes of `MPI_Send` and `MPI_Recv`. When sending a small message, the implementation may immediately copy the message to the Network Interface Card (NIC)[1] and return, marking the send operation as complete (see Figure 1(a)). While the actual transmission may still be in progress, the send buffer is safe for the application to use. In MPICH, this is called a lightweight send, which notably does not involve any *wait* blocks.

For larger messages, buffering costs can be significant. Instead, the message buffer pointer is passed to the NIC, which transmits the message directly from the buffer. `MPI_Send` must wait until the NIC signals completion, as the message buffer remains in use until then. This method, known as eager send mode, involves a single *wait* block (see Figure 1(b)).

When messages are even larger, early arrival at the receiver can cause issues, such as blocking the receiver's message queue or necessitating temporary buffer copying. To prevent unexpectedly receiving large messages, a handshake protocol is used: the sender sends a Ready to Send (RTS) message and waits for the receiver to post a matching buffer and reply with a Clear to Send (CTS) message. The sender then proceeds to send the message data similarly to eager mode. This rendezvous mode involves two *wait* blocks (see Figure 1(c)).

`MPI_Recv` operations vary as shown in Figure 1(d-f). Receiving an eager message, including those sent via lightweight send, involves a single *wait* block regardless of whether the message arrives before or after `MPI_Recv`. Receiving a rendezvous message requires two *wait* blocks.

Additional message modes with more complex protocols, such as pipeline mode, may involve multiple *wait* blocks. In pipeline mode, a large message is divided into chunks, and the implementation may control the number of concurrent chunks, leading to an indeterminate number of *wait* blocks.

B. Nonblocking and Asynchronous Task Patterns

The *wait* blocks in Figure 1 illustrate why `MPI_Send` and `MPI_Recv` are considered blocking operations. By focusing on the *wait* blocks in the block diagram, we can abstract tasks such as `MPI_Send` and `MPI_Recv` into three patterns, as shown in Figure 2: tasks that do not *wait*, tasks that contain a single *wait* block, and tasks with multiple *wait* blocks. During a *wait* block, the task is executed on a hardware device such as a NIC or GPU, within the OS kernel, or within a separate execution context such as a thread or process.

The *wait* block is often implemented as a busy poll loop, which wastes CPU cycles while the offloaded task is still in progress. Conversely, if the offloaded task finishes and the completion event is not immediately polled and acted upon, it can delay subsequent dependent work, adding latency to the workflow.

Rather than immediately waiting for an asynchronous task to complete, a program can, in principle, perform other jobs

[1]Here "NIC" loosely refers to either hardware operations or software emulations.

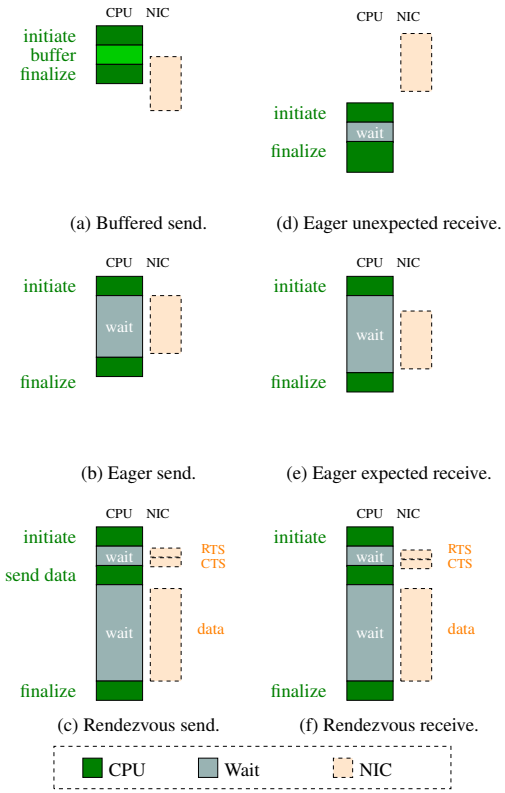

Fig. 1. Common communication modes: (a) Buffered eager send; (b) Normal eager send; (c) Rendezvous send; (d) Receiving an eager message that arrived before posting the receive; (e) Receiving an eager message that arrived after posting the receive; (f) Receiving a rendezvous message.

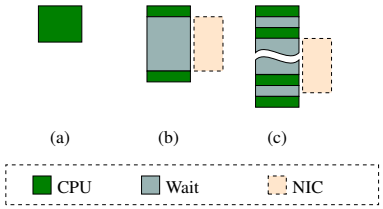

Fig. 2. Asynchronous Task patterns: (a) A task with no blocking parts; (b) A task with a single blocking part; (c) A task with multiple blocking parts.

that do not depend on the pending task. This is the idea behind MPI's nonblocking APIs. A nonblocking operation splits a corresponding blocking operation into two parts: starting and completion. For example, `MPI_Send` is divided into `MPI_Isend` and `MPI_Wait`.

Figure 3 illustrates how blocking patterns in Figure 2 are split into nonblocking patterns. If the task does not contain any *wait* blocks (Figure 3(a)), the split into a nonblocking pattern is somewhat arbitrary, but typically the entire operation is completed in the starting call, and the completion call will return immediately. If the task contains a single *wait* block (Figure 3(b)), it is naturally split just before the *wait* block. For tasks with multiple *wait* blocks (Figure 3(c)), the split

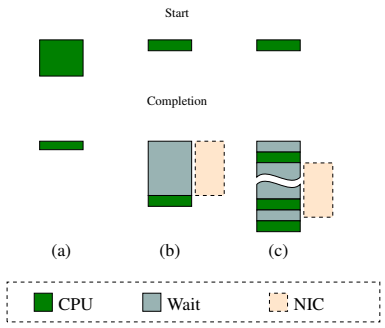

Fig. 3. Nonblocking task patterns: (a) A task with no blocking parts; (b) A task with a single blocking part; (c) A task with multiple blocking parts.

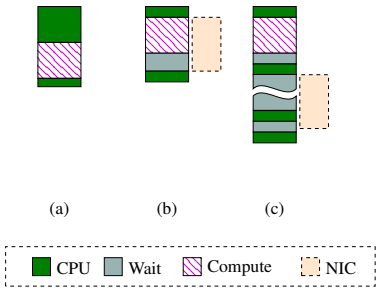

Fig. 4. Computation/communication overlap: (a) Communication with no blocking parts; (b) Communication with single blocking part; (c) Communication with multiple blocking parts.

occurs before the first *wait*. Generally, the starting call should avoid any *wait* blocks to preserve the nonblocking semantics.

Viewing MPI operations through the lens of *wait* patterns generalizes MPI nonblocking operations to common asynchronous programming patterns. For instance, the async/await syntax [9] in some programming languages provides a concise method to describe the *wait* patterns in a task. Event-driven programming [6], on the other hand, expresses the code following the *wait* block as event callbacks. In MPI, these async patterns are opaque, making MPI progress management obscure.

C. Computation/Communication Overlap

One of the primary goals of using nonblocking MPI operations is to achieve overlap between computation and communication. Ideally, the CPU cycles spent in a wait loop should instead be used for computation, enhancing overall efficiency. However, achieving this overlap with MPI is not straightforward. The concept of computation/communication overlap is illustrated in Figure 4. Immediately after initiating a nonblocking operation, the program enters a computation phase while the message data transmission is handled by the NIC hardware or another offloading device. Once the computation phase completes, the program resumes the *wait* for the nonblocking operation. If the communication has finished by then, the final *wait* returns immediately; otherwise, the *wait* time is significantly reduced. This overlap maximizes

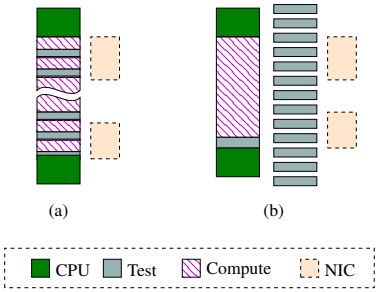

Fig. 5. Remedies for the lack of progress: (a) Intersperse progress tests inside computations; (b) Use a dedicated thread to continuously poll progress.

efficiency, improving overall performance and reducing time to solution.

The ideal overlap can be easily achieved for the case in Figure 4(b), where a single *wait* in the nonblocking operation allows for effective overlap. In contrast, Figure 4(a) shows a scenario with no *wait* block to save, offering no additional overlap compared to the blocking case. Converting a blocking operation without a *wait* into a nonblocking one only introduces overhead due to the creating and finalizing of a task handle (i.e., an MPI request). However, in most MPI implementations, this overhead is negligible. The situation is more complex in Figure 4(c), where multiple *wait* blocks are present, and computation only overlaps with the first *wait* block. This initial overlap is often insignificant compared to the total combined *wait* time. For instance, in a simple rendezvous message, the first *wait* involves waiting for a small protocol handshake, while the bulk of the message transmission occurs during the second *wait*. As a result, the opportunity for significant overlap is missed in such cases.

D. Role of Progress

The key issue with the scenario depicted in Figure 4(c) is the lack of progress. After the first *wait* ends, a small block of code needs to run to initiate the second *wait*. For asynchronous tasks with multiple *wait* blocks, this small block of code after each *wait* block must run to trigger the subsequent asynchronous tasks for the next *wait*. Polling for completion events and running the handlers to initiate the following asynchronous tasks constitute progress. Without adequate progress, the next steps in the asynchronous task are delayed, resulting in degraded performance.

There are two remedies for this lack of progress. One is to intersperse `MPI_Test` calls within the computation, as illustrated in Figure 5(a). However, this approach has at least three drawbacks. First, breaking the computation into parts and interspersing it with progress calls significantly increases code complexity and may not always be feasible. For example, the computation might be encapsulated in an opaque function, or the bulk of it might be spent in a math routine from an external library. Second, if progress polling is too frequent, many polls will waste CPU cycles without benefit, and the context switching between computation and progress polling adds overhead. Consequently, frequent polling decreases performance. Third, if progress polling is too sparse, the likelihood of polling just after a communication step completes is low, resulting in imperfect computation/communication overlap.

An alternative solution is to use a dedicated CPU thread for polling progress. This approach ensures sufficient progress and maximizes the overlap between computation and communication. However, it also wastes CPU cycles when a communication step is not ready and occupies an entire CPU core. While modern HPC systems often have many cores, dedicating a CPU core for progress can be acceptable for some applications. However, this becomes problematic when multiple processes are launched on a single node. If each process has its own progress thread, it can quickly exhaust CPU cores and severely impact performance. Additionally, applications may use other asynchronous subsystems besides MPI, each potentially requiring its own progress thread, leading to competition for limited core resources.

E. Managing MPI Progress

Implementing a progress thread with MPI is challenging, primarily because MPI does not provide explicit APIs for invoking MPI progress. MPI progress is largely hidden within the implementation, with the assumption that a system-optimized MPI will provide strong progress, thereby reducing the need for explicit progress. However, as discussed earlier, this assumption may not always hold true. Calling any MPI function may or may not invoke MPI progress, and when it does, it may not serve global progress. For instance, calling `MPI_Test` on one MPI request may not necessarily advance other MPI requests. Furthermore, any MPI function that invokes progress may contend for locks with another thread calling MPI functions. Managing MPI progress can feel almost magical when it works, but extremely frustrating when it fails.

One of the more explicit ways to invoke MPI progress is by calling `MPI_Test` (or any of its variants). `MPI_Test` requires an explicit MPI request parameter. However, a dedicated progress thread is often isolated from the context that initiates the actual MPI operations, making it challenging to synchronize MPI requests between the computation threads and the progress thread. This synchronization of MPI request objects imposes a significant burden on a many-task system design. Therefore, to enable applications to build effective progress engines, MPI needs to provide mechanisms for invoking progress independent of specific MPI requests.

F. Collating Progress

In addition to the inconvenience of having to use an MPI request to call `MPI_Test`, polling progress for individual MPI requests is inefficient. It is more efficient to poll for all events, process them one by one, and then check whether a specific MPI request has been completed from the event handling. This is referred to as collating progress. Collating progress ensures that all parts of the program are progressing towards completion, rather than waiting unnecessarily for each individual operation to be completed sequentially. Collating

progress can help improve overall application performance by reducing bottlenecks and enhancing concurrency.

In addition to collating progress for network operations, an MPI library internally needs to manage the progress of multiple asynchronous subsystems. For example, data transfer may involve GPU device memory, meaning a conventional MPI send and receive could include asynchronous memory copy operations between host and device memory. MPI-IO may introduce asynchronous storage I/O operations. Collectives are often implemented as a series of nonblocking point-to-point communications following a multi-stage pattern similar to Figure 3(c). Additionally, MPI communication may internally utilize different subsystems depending on whether the communication is between on-node processes or inter-node processes. All these asynchronous subsystems require progress, and it is often more convenient and efficient to collate them.

Listing 1 shows the pseudocode of MPICH's internal progress function.

```
int MPIDI_progress_test(MPID_Progress_state *state) {
    int mpi_errno = MPI_SUCCESS, made_progress = 0;

    /* asynchronous datatype pack/unpack */
    Datatype_engine_progress(&made_progress);
    if (made_progress) goto fn_exit;

    /* collective algorithms */
    Collective_sched_progress(&made_progress);
    if (made_progress) goto fn_exit;

    /* intranode shared memory communication */
    Shmem_progress(&made_progress);
    if (made_progress) goto fn_exit;

    /* internode netmod communication */
    Netmod_progress(&made_progress);
    if (made_progress) goto fn_exit;

  fn_exit:
    return mpi_errno;
}
```

Listing 1. Pseudocode for MPICH's progress function

This progress routine is called whenever an MPI function requires progress. Collating progress assumes that the cost is negligible if a subsystem has no pending tasks. For the datatype engine, collective, and shared memory (shmem) subsystems, an empty poll incurs a cost equivalent to reading an atomic variable. However, this is not always the case with netmod progress, so we place netmod progress last and skip it whenever progress is made with other subsystems. Additionally, MPICH's progress function accepts a state variable from the calling stack, providing an opportunity for the caller to tune the progress performance according to the context. For example, from a context where only netmod progress is needed, the progress state can be set to skip progress for all other subsystems. Since MPI implementations already perform collated event-based progress internally, exposing MPI progress as an explicit API for applications is straightforward.

G. Case for Interoperable MPI Progress

As we have discussed, the patterns of MPI internal operations are similar to those of general asynchronous tasks that applications may create. Therefore, the design and optimization of MPI progress should be applicable to application-layer asynchronous tasks as well. In fact, current MPI implementations already handle several async subsystems internally, making it straightforward to extend this capability to work with external tasks. We refer to this concept as "interoperable MPI progress."

Interoperable MPI progress provides applications with a mature progress engine, eliminating the need to create and maintain separate progression mechanisms for each new async system. Additionally, integrating user-layer progress within MPI progress is more convenient and efficient.

Another advantage of interoperable MPI progress is that it allows for the implementation and extension of MPI subsystems at the user level. For instance, users could implement collectives in user space by adding a progress hook into MPI's progress, similar to the `Collective_sched_progress` in Listing 1. This approach promotes a modular design where parts of MPI are built on top of a core MPI implementation, enhancing both flexibility and stability. Furthermore, a core MPI set that facilitates the building of MPI extensions can stimulate broader community research activities and infuse new life into MPI.

III. MPICH Extensions For Interoperable Progress

In this section, we present new extension APIs developed in MPICH that enable applications to more effectively manage MPI progress and to extend MPI through interoperable MPI progress.

A. MPIX Streams

First, we refresh the MPIX Stream extension introduced in our previous work [8]. An MPIX Stream represents an internal communication context within the MPI library, defined as a serial execution context. All operations attached to an MPIX Stream are required to be issued in a strict serial order, eliminating the need for lock protection within the MPI library. The default stream, `MPIX_STREAM_NULL` can be used wherever MPIX Stream is needed in the interface, thus, MPIX Stream is not a prerequisite. However, exposing MPIX Stream allows applications to influence and control MPI progress to be more targeted and to avoid contention, addressing one of the key performance factors in a multi-threaded application.

An MPIX Stream is created using the following API:

```
int MPIX_Stream_create(MPI_Info info, MPIX_Stream *
    stream)
```

To use an MPIX Stream in MPI communications, you must first create a stream communicator with the following function:

```
int MPIX_Stream_comm_create(MPI_Comm parent_comm,
    MPIX_Stream stream, MPI_Comm *stream_comm)
```

A stream communicator can be used the same way as a conventional MPI communicator, except that all operations on a stream communicator will be associated with the corresponding MPIX Stream context. While an MPIX Stream is naturally suited for a thread context, it can also be applied to any semantically serial construct. For example, the serial context can be manually enforced through thread barriers, or originate from a specific runtime such as a CUDA stream. Info hints offer a flexible mechanism for implementations to extend support and apply specific optimizations. For more detailed information on MPIX Streams, please refer to our previous work[8].

B. Explicit MPI Progress

To address the need for making MPI progress without being tied to specific MPI requests, We propose an API that allows applications to advance MPI progress for a specific MPIX Stream:

```
int MPIX_Stream_progress(MPIX_Stream stream)
```

An explicit MPIX Stream can be used to avoid unnecessary thread contention or to influence how each subsystems are being progressed. For example, in Listing 1, hints can be provided to the MPIX Streams to skip some progress components if the subsystem does not require them.

C. MPIX Async Extension

`MPIX_Stream_progress` allows applications to incorporate MPI progress into their progression schemes. However, to make MPI progress truly interoperable, we also need a mechanism for applications to add progress hooks into the MPI progress system. This is accomplished with the following extension:

```
int MPIX_Async_start(MPIX_Async_poll_function
    poll_fn, void *extra_state, MPIX_Stream stream
    )
```

The `poll_fn` parameter is a user-defined progress hook function that is called from within MPI progress (e.g., inside `MPIX_Stream_progress` or `MPI_Test`) along with MPI's internal progress hooks (See Listing 1). `extra_state` is a user-defined handle or a state pointer that will be passed back to `poll_fn`. The `stream` parameter attaches the task to the corresponding MPIX Stream, including the default stream, `MPIX_STREAM_NULL`. `poll_fn` has the following signature:

```
typedef struct MPIR_Async_thing *MPIX_Async_thing;
typedef int (MPIX_Async_poll_function)(
    MPIX_Async_thing);
```

An opaque `struct` pointer, `MPIX_Async_thing`, is used instead of directly passing `extra_state` back to `poll_fn`. This provides some flexibility for implementations to support additional features. `MPIX_Async_thing` combines application-side context (i.e., `extra_state`) and the implementation-side context. Inside `poll_fn`, the original `extra_state` can be readily retrieved with:

```
void *MPIX_Async_get_state(MPIX_Async_thing
    async_thing)
```

`poll_fn` returns either `MPIX_ASYNC_PENDING` if the async task is in progress or `MPIX_ASYNC_DONE` if the async task is completed. Before `poll_fn` returns `MPIX_ASYNC_DONE`, it must clean up the application context associated with the async task, typically by freeing the structure behind `extra_state`. The MPI library will then free the context behind `MPIX_Async_thing`.

The MPIX Async interface allows users to extend MPI's functionality and integrate custom progression schemes into MPI progress. For example, an MPI collective can be viewed as a fixed task graph composed of individual operations and their dependencies. By defining `poll_fn`, one can advance a specific task graph for a custom collective algorithm within MPI progress. Integrating into MPI progress simplifies the process by eliminating the need for constructing separate progress mechanisms and avoiding performance issues such as managing MPI request objects, extra progress threads, and thread contention.

MPIX Async is not a rehash of MPI Generalized Request. Rather they can be used together and complement each other. MPIX Async provides a progress mechanism, while MPI Generalized Request provides the handle representation. Together, they allow users to create nonblocking tasks that not only can be queried via MPI request API but also can be progressed using the same MPI progression scheme.

D. Completion Query on MPI Requests

When a task finishes its computation for a given stage, it must wait for the completion of its dependent nonblocking operations. However, `MPI_Wait` may also perform progress in addition to local queries and wait blocks. This is good when `MPI_Wait` is the main progression mechanism. However, when there is a separate progress engine, invoking redundant progress outside the progress engine wastes CPU cycles, creates thread contentions, and degrades performance.

The following API provides a pure request completion query function that does not have the side effects of triggering progress:

```
bool MPIX_Request_is_complete(MPI_Request request)
```

The implementation simply queries an atomic flag for the request, resulting in minimal overhead when repeatedly polling this function. Importantly, there are no side effects that would interfere with other requests or other progress calls. `MPIX_Request_is_complete` is also useful in the MPIX Async `poll_fn` when the application-layer task is built upon MPI operations. Each MPI progress may internally use a context to coordinate various parts, thus, invoking progress recursively inside the `poll_fn` is prohibited.

IV. Examples And Evaluations

In this section, we present various examples using the extensions introduced in the last section and demonstrate

various performance factors in MPI progress. An important metric for quantifying progress performance is the progress latency, defined as the average elapsed time between a task's completion and when the user code responds to the event. We use a dummy task created with MPIX Async interface to directly measure progress latency. Unless otherwise noted, our experiments were conducted on a local workstation with an 8-core i7-7820X CPU.

A. Dummy task

For most of the following examples, we use a dummy task that completes after a predetermined duration. This simulates an asynchronous job completed via offloading. Instead of querying a completion status, we check for the elapsed time. The code is provided in Listing 2. By presetting the duration for the dummy task to complete, we can measure the latency of the progress engine's response to the completion event.

```
#define TASK_DURATION 1.0
#define NUM_TASKS 10

struct dummy_state {
    double wtime_finish;
    int *counter_ptr;
};

void add_stat(double latency);  /* impl. omitted */
void report_stat(void);         /* impl. omitted */

static int dummy_poll(MPIX_Async_thing thing) {
    struct dummy_state *p = MPIX_Async_get_state(thing);
    double wtime = MPI_Wtime();
    if (wtime >= p->wtime_finish) {
        add_stat(wtime - p->wtime_finish) * 1e6;
        (*(p->counter_ptr))--;
        free(p);
        return MPIX_ASYNC_DONE;
    }
    return MPIX_ASYNC_NOPROGRESS;
}

static void add_async(int *counter_ptr) {
    struct dummy_state *p = malloc(sizeof(struct
        dummy_state));
    p->wtime_finish = MPI_Wtime() + TASK_DURATION;
    p->counter_ptr = counter_ptr;
    MPIX_Async_start(dummy_poll, p);
}

int main(int argc, const char **argv) {
    MPI_Init(NULL, NULL);

    int counter = NUM_TASKS;
    for (int i = 0; i < NUM_TASKS; i++) {
        add_async(&counter);
    }

    /* progress */
    while (counter > 0) {
        MPIX_Stream_progress(MPIX_STREAM_NULL);
    }

    report_stat();
    MPI_Finalize();
    return 0;
}
```

Listing 2. An example of dummy async task using MPIX Async extensions with synchronization counter, wait-progress loop, and stubs for latency benchmarking

Since there is no real computation work in our example, we spin `MPIX_Stream_progress` after adding the tasks in the same thread. Because this progress does not take individual task handles, it can be easily moved into a progress thread or a progress engine, achieving computation overlap.

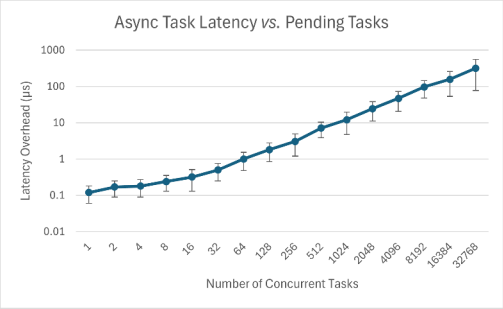

Fig. 6. Latency overhead in microseconds as the number of pending async tasks increases.

B. Performance Factors in Async Progress

Several performance factors influence the average response latency to event completion in asynchronous progress.

1) Number of pending tasks: During each MPI progress call (e.g., `MPIX_Stream_progress`), MPI will invoke the async `poll_fn` for each pending async task sequentially. The cycles spent processing numerous tasks may delay the response time to a specific task's completion event. Therefore, as the number of pending tasks increases, we expect an increase in response latency. This expectation is confirmed by the experimental results shown in Figure 6.

If all the pending tasks are independent, each progress call must invoke `poll_fn` for every pending task, leading to a performance degradation as the number of pending tasks rises. Notably, when there are fewer than 32 pending tasks, the latency overhead remains below 0.5 microseconds.

Most applications do not create thousands of independent tasks randomly. Typically, tasks have dependencies on each other, forming a task graph, or they are grouped into streams with implicit linear dependencies. When tasks have dependencies, it is possible to skip polling progress for tasks whose dependent tasks are not yet completed. While implementing a general-purpose task management system to track dependencies is complex, we will demonstrate in Section IV-C through an example how users can manage task dependencies within their `poll_fn`.

2) Poll Function Overhead: In addition to the number of pending tasks, the overhead of individual progress poll functions (`poll_fn`) can also affect the average event response latency. If a single `poll_fn` takes a disproportionate amount of time to execute, the overall response time to events will increase. This effect is illustrated in Figure 7, where we manually inserted delays in `poll_fn` when the task is still pending.

The MPIX Async interface is designed for lightweight `poll_fn` functions and is not suitable for tasks that require significant CPU cycles to respond to an event. This is generally true for all collated progress mechanisms: when one part of

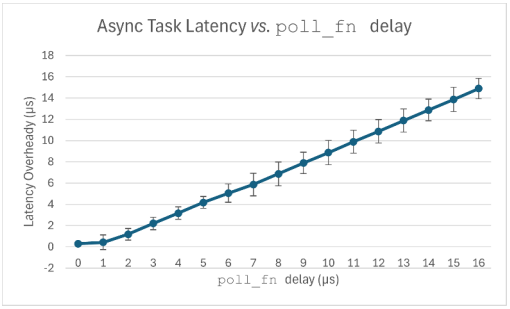

Fig. 7. Impact of poll function overhead on event response latency. Each measurement runs 10 concurrent pending tasks. The delay is implemented by busy-polling `MPI_Wtime`.

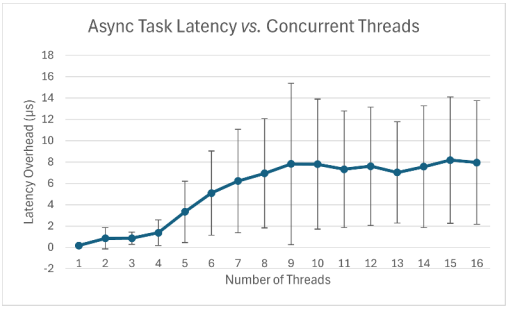

Fig. 8. Latency overhead in microseconds as the number of concurrent progress threads increases. Each measurement runs 10 concurrent pending tasks.

the progress takes significant overhead, it negatively impacts the performance of other tasks.

To avoid heavy `poll_fn` overhead, it is recommended to enqueue events and postpone the heavy work outside of the progress callbacks. This approach ensures that the `poll_fn` remains lightweight, minimizing its impact on overall performance.

3) Thread Contention: When multiple threads concurrently execute progress, they contend for a lock to avoid corrupting the global pending task list. Even if the tasks are independent, multiple threads running collated progress will still contend for locks, leading to performance degradation. As illustrated in Figure 8, the observed latency increases with the number of concurrent progress threads.

It is important to note that individual `poll_fn` functions may access application-specific global states that require lock protection from other parts of the application code. This lock protection should be implemented within the `poll_fn` by the application. MPI only ensures thread safety between MPI progress calls.

To avoid performance degradation, it is advisable to limit the number of progress threads—a single progress thread often suffices. Sometimes, MPI progress loops are invoked implicitly. For example, blocking calls in MPI, such as `MPI_Recv`, often include implicit progress similar to `MPI_Wait`. Even initiation calls, such as `MPI_Isend`, may contend for a lock with the MPI progress thread. This global lock contention contributes to the notorious poor performance of `MPI_THREAD_MULTIPLE` [8].

Using MPIX Stream appropriately can mitigate the issue of global thread contention. We will demonstrate this in Section IV-D.

C. Async task class

To avoid wasting cycles polling progress for tasks with pending dependencies, it is essential to track task dependencies. This can be managed either within MPI implementations or inside the callback `poll_fn`. However, handling general task graphs inside MPI implementations needs to handle unbounded complexity, while applications often have specific and much simpler dependency structures. Therefore, it is more

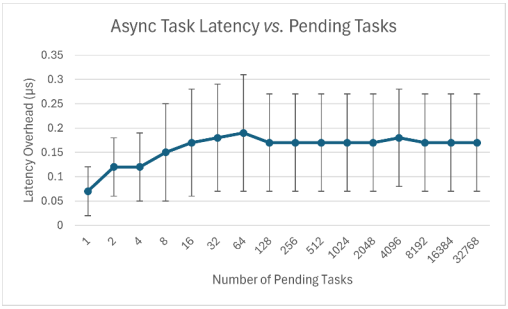

Fig. 9. Latency versus the number of pending tasks when the progress callback only checks the task at the top of the queue.

effective for applications to manage task dependencies within the `poll_fn`. For example, instead of polling progress for individual asynchronous tasks, users can design task subsystems or asynchronous task classes. Within the `poll_fn`, they can poll progress for the entire task class. When the tasks are topologically sorted according to dependency, the `poll_fn` only needs to poll the head tasks rather than to poll every task.

```
for (int i = 0; i < count; i++) add_async(i);
MPIX_Async_start(class_poll, task_graph);
```

Listing 3. Use a single `poll_fn` to manage a class of async tasks.

Shown in Listing 3, a single `MPIX_Async_start` is used to register progress for all tasks within the class. Individual tasks are launched in `add_async` function by adding to a separately managed task graph (omitted in Listing 3). The `class_poll` function will return `MPIX_ASYNC_NOPROGRESS` until finalization. Effectively, we are adding a custom progress component in Listing 1.

When all the tasks in the task class follow a linear dependency, the `poll_fn` only needs to poll a single task each time. As shown in Figure 9, the average latency stays constant (within measurement noise) regardless of the number of pending tasks.

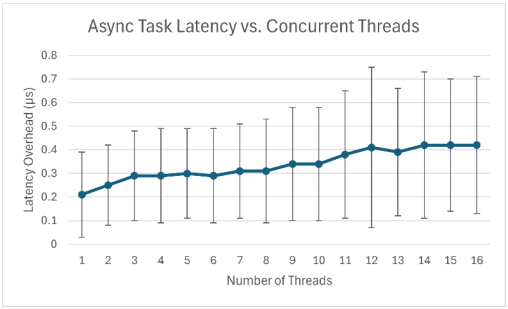

Fig. 10. Latency versus the number of concurrent progress threads using different MPIX streams. Each measurement runs 10 concurrent pending tasks.

D. Concurrent progress streams

If running multiple progress threads is necessary, explicit MPIX Stream can be used to avoid contentions between the various threads. To demonstrate, we create separate MPIX streams and have each thread use its own stream in `MPIX_Async_start` and `MPIX_Stream_progress`. The result is shown in Figure 10. In contrast to Figure 5, the average progress latency does not increase significantly as the number of threads increases when separate MPIX streams are used. The slight increase in latency is within measurement noise and is attributable to core power fluctuations due to the number of active cores.

E. User-level collective algorithms

One of the motivations behind the MPIX Async extension is to enable user-level implementation of collective algorithms with performance comparable to native implementations. Collective algorithms are essentially a collection of communication patterns built on a core set of operations, including MPI point-to-point operations, buffer copies, and local reductions.

A key advantage of native collective implementations is their tight integration with the MPI progress. MPIX Async is designed to provide the same level of integration to user-level applications.

In Listing 4, we present an example of implementing a user-level allreduce algorithm using the MPIX Async APIs. This example implements the recursive doubling allreduce algorithm[10]. For simplicity, the datatype is restricted to `MPI_INT`, the op to `MPI_SUM`, and the number of processes to a power of 2.

To compare the performance of this custom user-level allreduce against MPICH's `MPI_Iallreduce` using the same recursive doubling algorithm, we conducted experiments on the Bebop cluster at Argonne National Laboratory Computing Resource Center (LCRC). The experiment measures the latency of allreduce of a single integer. The results are shown in Figure 11.

The custom user-level implementation actually outperforms MPICH's native `MPI_Iallreduce`. We believe this is due to the specific assumptions and shortcuts in the custom implementation. For example, we assume the number of processes is a power of 2 and that `sendbuf` is `MPI_INPLACE`, which avoids certain checks and branches. Additionally, restricting to `MPI_INT` and `MPI_SUM` avoids a datatype switch and the function-call overhead of calling an operation function. This highlights an advantage of custom code over an optimized MPI implementation: the former can leverage specific contexts from the application to avoid complexities and achieve greater efficiency.

```
struct myallreduce {
    int *buf, *tmp_buf;
    MPI_Comm comm;
    int rank, size, tag, mask, count;
    MPI_Request reqs[2];  /* send & recv requests */
    bool *done_ptr; /* external completion flag */
};

static int my_poll(MPIX_Async_thing thing) {
    struct myallreduce *p = MPIX_Async_get_state(thing);
    int req_done = 0;
    for (int i = 0; i < 2; i++) {
        if (p->reqs[i] == MPI_REQUEST_NULL) {
            req_done++;
        } else if (MPIX_Request_is_complete(p->reqs[i])) {
            MPI_Request_free(&p->reqs[i]);
            req_done++;
        }
    }
    if (req_done != 2) {return MPIX_ASYNC_NOPROGRESS;}
    if (p->mask > 1) {
        for (int i = 0; i < p->count; i++)
            p->buf[i] += p->tmp_buf[i];
    }
    if (p->mask == p->size) {
        *(p->done_ptr) = true;
        free(p->tmp_buf);  free(p);
        return MPIX_ASYNC_DONE;
    }
    int dst = p->rank ^ p->mask;
    MPI_Irecv(p->tmp_buf, p->count, MPI_INT, dst, p->tag, p
        ->comm, &p->reqs[0]);
    MPI_Isend(p->buf, p->count, MPI_INT, dst, p->tag, p->
        comm, &p->reqs[1]);
    p->mask <<= 1;

    return MPIX_ASYNC_NOPROGRESS;
}

void MyAllreduce(const void *sendbuf, void *recvbuf, int
     cnt, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) {
    int rank, size;
    MPI_Comm_rank(comm, &rank); MPI_Comm_size(comm, &size);

    /* only deal with a special case */
    assert(sendbuf == MPI_IN_PLACE && datatype == MPI_INT
        && op == MPI_SUM && is_pof2(size));

    struct myallreduce *p = malloc(sizeof(*p));
    p->buf = recvbuf; p->count = cnt;
    p->tmp_buf = malloc(cnt * sizeof(int));
    p->reqs[0] = p->reqs[1] = MPI_REQUEST_NULL;
    p->comm = comm;
    p->rank = rank; p->size = size;
    p->mask = 1; p->tag = MYALLREDUCE_TAG;

    bool done = false;
    p->done_ptr = &done;

    MPIX_Async_start(my_poll, p, MPIX_STREAM_NULL);
    while (!done) MPIX_Stream_progress(MPIX_STREAM_NULL);
}
```

Listing 4. An example of a user-level allreduce algorithm

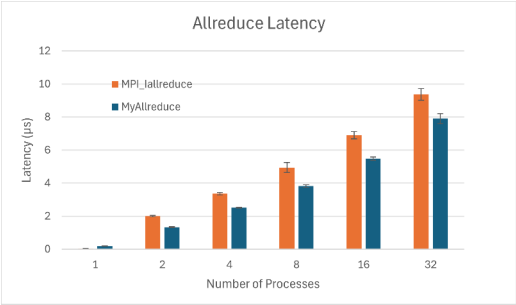

Fig. 11. Custom single-integer allreduce latency vs `MPI_Iallreduce`. Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz. Nodes interconnect via Omni-Path Fabric. One process per node.

V. Related Work

Several related works address issues with MPI progress and the management of asynchronous tasks within MPI. Here, we provide a brief review and compare these approaches with our proposed methods.

A. MPICH Asynchronous Progress

With MPICH and its derived MPI implementations, users can set an environment variable, to enable a dedicated background thread to poll progress in a busy loop. This is the simplest method for ensuring that MPI will always progress and complete any nonblocking operations after they are initiated, without requiring additional MPI calls from the application. However, this method has significant performance issues and is generally not recommended. MVAPICH [11] has proposed a design to address these issues by identifying scenarios where asynchronous progress is required and putting the async thread to sleep when it is not required or beneficial. Their benchmarks showed up to a 60% performance improvement over the original MPICH design.

With the extension `MPIX_Stream_progress`, applications can easily implement the async progress thread within the application layer. The same tuning design as that adopted in MVAPICH can be applied. Instead of the implementation detecting where async progress is required, the application can know where it is needed by design, thus controlling the progress thread more precisely and directly. `MPIX_Stream_progress` also allows applications to use MPIX streams to target async progress for specific contexts, thereby avoiding lock contention issues altogether. In contrast, it is difficult for an MPI implementation to detect user contexts or determine the scope of global progress.

Casper [12] is a process-based asynchronous progress approach for MPI. It leverages one or more auxiliary "ghost" processes to offload communication progress for all application processes on a node. The Casper approach addresses the contention and oversubscription issues of progress threads and shows good benefits for passive-target RMA applications. Still, progress for passive-target RMA is not always straightforward using standard MPI APIs, which ultimately adds complexity to the Casper implementation. `MPIX_Stream_progress` can enable progress explicitly on specific resources, such as RMA windows, thereby keeping the code simple and potentially more efficient by avoiding unnecessary global progress.

B. MPIX Schedule

MPIX Schedule[5] is a proposal to expose MPI's internal nonblocking collective system to applications so that users can create custom nonblocking collective algorithms or a series of coordinated MPI operations similar to a nonblocking MPI collective. The proposal only has APIs to add operations represented as an MPI request or an MPI op, thus it would be difficult to extend to non-MPI operations such as GPU asynchronous copy and offloading tasks. More critically, the lack of a progress mechanism limits the performance users can achieve compared to what is possible within an MPI implementation.

In contrast, the MPIX Async APIs address the root issue of providing interoperable MPI progress. Its simple yet flexible progress hook-based interface can accommodate arbitrary asynchronous tasks. Combined with generalized requests, it enables users to effectively experiment with MPI extensions, such as experimental collective algorithms, entirely from the application layer.

C. MPIX Continue

MPIX Continue [4] is a proposal that allows MPI to call back a user-defined function upon the completion of a request, eliminating the need for the application to continuously test the request for completion. The motivation behind the MPIX Continue APIs is to simplify the management of MPI requests within a task-based programming model.

Our proposed extensions address many of the challenges that the MPIX Continue proposal seeks to resolve. `MPIX_Stream_progress` can be used to implement a polling service that integrates with the task runtime system without requiring synchronization of MPI requests. Meanwhile, `MPIX_Request_is_complete` allows tasks to query the status of their dependent requests without triggering redundant progress invocations.

VI. Summary

In summary, we present a suite of MPI extensions crafted to provide an interoperable MPI progress, which grants applications explicit control over MPI progress management, the ability to incorporate user-defined asynchronous tasks into MPI, and seamless integration with task-based and event-driven programming models. The `MPIX_Stream_progress` and `MPIX_Request_is_complete` APIs effectively decouple the context for invoking MPI progress and querying the completion status of MPI requests, thereby circumventing synchronization complexities and task-engine interference. Meanwhile, the MPIX Async proposal empowers applications to integrate custom progress hooks into MPI progress, enabling them to harness MPI progress and extend MPI functionality from the user layer. Through examples and micro-benchmark

testing, we demonstrate the effectiveness of these extensions in bringing MPI to modern asynchronous programming.

ACKNOWLEDGMENT

This research was supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

REFERENCES

[1] E. Castillo, N. Jain, M. Casas, M. Moreto, M. Schulz, R. Beivide, M. Valero, and A. Bhatele, "Optimizing computation-communication overlap in asynchronous task-based programs," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 380–391.

[2] M. Sergent, M. Dagrada, P. Carribault, J. Jaeger, M. Pérache, and G. Papauré, "Efficient communication/computation overlap with MPI+OpenMP runtimes collaboration," in *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings 24*. Springer, 2018, pp. 560–572.

[3] D. J. Holmes, A. Skjellum, and D. Schafer, "Why is MPI (perceived to be) so complex?: Part 1 – does strong progress simplify MPI?" in *Proceedings of the 27th European MPI Users' Group Meeting*, 2020, pp. 21–30.

[4] J. Schuchart, P. Samfass, C. Niethammer, J. Gracia, and G. Bosilca, "Callback-based completion notification using MPI continuations," *Parallel Computing*, vol. 106, p. 102793, 2021.

[5] D. Schafer, S. Ghafoor, D. Holmes, M. Ruefenacht, and A. Skjellum, "User-level scheduled communications for MPI," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019, pp. 290–300.

[6] P. Haller and M. Odersky, "Event-based programming without inversion of control," in *Joint Modular Languages Conference*. Springer, 2006, pp. 4–22.

[7] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, 1999, pp. 23–32.

[8] H. Zhou, K. Raffenetti, Y. Guo, and R. Thakur, "MPIX Stream: An explicit solution to hybrid MPI+X programming," in *Proceedings of the 29th European MPI Users' Group Meeting*, 2022, pp. 1–10.

[9] S. Okur, D. L. Hartveld, D. Dig, and A. v. Deursen, "A study and toolkit for asynchronous programming in C#," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1117–1127.

[10] M. Ruefenacht, M. Bull, and S. Booth, "Generalisation of recursive doubling for allreduce: Now with simulation," *Parallel Computing*, vol. 69, pp. 24–44, 2017.

[11] A. Ruhela, H. Subramoni, S. Chakraborty, M. Bayatpour, P. Kousha, and D. K. Panda, "Efficient design for MPI asynchronous progress without dedicated resources," *Parallel Computing*, vol. 85, pp. 13–26, 2019.

[12] M. Si, A. J. Peña, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa, "Dynamic adaptable asynchronous progress model for MPI RMA multiphase applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 1975–1989, 2018.