2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)

User-Level Scheduled Communications for MPI

Derek J. Schafer, K. Sheikh Ghafoor
Dept. of Computer Science
Tennessee Technological University
Cookville, TN, 38505, USA
Email: {djschafer42@students., sghafoor@}tntech.edu

Daniel J. Holmes
EPCC
University of Edinburgh
Edinburgh, Scotland, UK
Email: d.holmes@epcc.ed.ac.uk

Martin Rüfenacht, Anthony Skjellum
SimCenter &
Dept. of Computer Science and Engineering
University of Tennessee at Chattanooga
Chattanooga, TN, 37403, USA
Email: {martin-ruefenacht,tony-skjellum}@utc.edu

***Abstract*—Composability is one of seven reasons for the long-standing and continuing success of MPI. Extending MPI by composing its operations with user-level operations provides useful integration with the progress engine and completion notification methods of MPI. However, the existing extensibility mechanism in MPI (generalized requests) is not widely utilized and has significant drawbacks.**

MPI can be generalized via scheduled communication primitives, for example, by utilizing implementation techniques from existing MPI-3 nonblocking collectives and from forthcoming MPI-4 persistent and partitioned APIs. Non-trivial schedules are used internally in some MPI libraries; but, they are not accessible to end-users.

Message-based communication patterns can be built as libraries on top of MPI. Such libraries can have comparable implementation maturity and potentially higher performance than MPI library code, but do not require intimate knowledge of the MPI implementation. Libraries can provide performance-portable interfaces that cross MPI implementation boundaries. The ability to compose additional user-defined operations using the same progress engine benefits all kinds of general purpose HPC libraries.

We propose a definition for MPI schedules: a user-level programming model suitable for creating persistent collective communication composed with new application-specific sequences of user-defined operations managed by MPI and fully integrated with MPI progress and completion notification. The API proposed offers a path to standardization for extensible communication schedules involving user-defined operations. Our approach has the potential to introduce event-driven programming into MPI (beyond the tools interface), although connecting schedules with events comprises future work.

Early performance results described here are promising and indicate strong overlap potential.

I. Introduction

It is useful to extend MPI communication operations with new and composed operations that integrate with the progress engine and completion notification methods of MPI. But, generalized requests, defined in MPI-2 to support such extensions, are not fit-for-purpose and are rarely used in practice. The advent of persistent collective communication and forthcoming partitioned point-to-point communication opens an opportunity for a new, high performance solution.

The key contributions of this paper are as follows: MPI can be generalized by the introduction of scheduled communication primitives that utilize forthcoming MPI-4 persistent APIs. Non-trivial schedules are already used internally in some MPI libraries, but they are not accessible to end users as first-class programming constructs. These constructs will prove most useful when coupled with a strong progress engine and blocking completion notification. In many cases, MPI library writers will be able to add message-based collectives of comparable implementation maturity and potentially higher performance to those implemented internally in an MPI library, yet without intimate knowledge of the implementation or need to augment it internally. Those algorithms would provide a new kind of performance-portable MPI code that crosses MPI implementation boundaries. MPI schedules provide a user-level programming model suitable for creating persistent collective communication, nonblocking communication, plus new, application-specific sequences of operations in MPI. While schedules exist within some implementations of MPI, our approach provides a user-level API that integrates with MPI progress and completion notification. The API proposed offers a path to standardization for extensible communication sequences with the potential to introduce event-driven programming as a future extension to MPI (not just tools-related). Means for connecting scheduled communication with internal and external events is considered for future work.

The remainder of this paper is organized as follows: Section II describes the motivations and background for this work. Section III describes the design of user-level schedule communication including API specifications. Section IV describes qualitative performance modeling. Implementation issues and details, including how this work builds on the ExaMPI [1] research implementation of MPI, are described in Section V. Section VI presents some preliminary results with user-level schedules. Section VII describes possible extensions to user-level scheduled communication and indicates plans for future work, while Section VIII offers conclusions.

II. Background and Motivation

This section addresses background concepts and motivations for this paper, including how this fits with forthcoming additions to MPI, and where it builds on previous design, implementation, and/or standardization work.

A. Motivations

The following properties of MPI-4 (accepted and/or proposed functionality), inform this effort:

- Collective communication operations approved for MPI have initialization operations. Such INITs are currently collective (non-local, may synchronize) because of effects involving hand-off between the application and MPI of array parameters (e.g., displacements) to certain collectives plus the non-blocking properties of MPI_REQUEST_FREE. The future addition of local variants of these operations will not break backward compatibility, but will require a stronger completion form of MPI_REQUEST_FREE to ensure that resources held by MPI are relinquished to the user. Also, addition of local variants of currently non-local functionality must be considered throughout the MPI Standard (e.g., MPI_WIN_ICREATE, etc). The present semantic compromise is clearly shown in [2]; the Venn diagram(s) depicted there that show collective initialization procedures as incomplete and non-local whereas collective initiation procedures and point-to-point initialization procedures are both shown as *incomplete* and *local*.
- Channels—point-to-point communication between two MPI processes—will be achieved in MPI via partitioned communication [3], which is presently being considered for MPI-4. Partitioned communication, in the fullness of time, will have blocking, nonblocking, and persistent API variants for point-to-point, collective, one-sided, and I/O operations. Persistent partitioned communication adds a new concern to planning schedules (the key goal of this paper): the user may inform MPI that individual parts of an outgoing message are ready for transmission without being required to create and manage multiple messages. A similar semantic extension, the ability to consume individual parts of an incoming message without being required to create and manage multiple messages, is also being considered.
  Breaking an MPI operation into pieces that are scheduled separately is not novel. However, allowing the user to indicate when some of the pieces are ready, as proposed for partitioned communication, is new and it presents a new challenge for planning schedules. Naively, partitioning could be scheduled in the same manner as separate operations, that is, one operation per partition. In that case, the completion of the whole partitioned operation is achieved when all the separate per-partition operations have completed - much like a call to MPI_WAITALL but without all the MPI_REQUEST objects. Benchmarking work for partitioned communication [3] shows that the separate messages approach is less efficient than a straightforward implementation of the partitioned approach. Less naively, user-driven execution of parts of a schedule comprises a DAG schedule with multiple root vertices (one per partition), where each root vertex requires a user down-call to satisfy its input dependencies. This seems to present a major challenge to the linearization-into-command-queue technique used, for example, in LibNBC [4] and in Open MPI [5]. Any technique to store an execution plan or schedule for MPI operations is likely to have to take this new consideration into account - requiring structural changes that acknowledge the new sources of input dependencies between the scheduled sub-tasks.
- Some communication patterns involve multiple persistent requests and managing these requests could become cumbersome. For example, a user might want to perform entire communication protocols that are not defined by MPI (e.g., halo exchange for domain decomposition) with an all-reduce (e.g., for monitoring conservation of energy/mass) plus another all-reduce (e.g., for termination criteria) before finishing off with collective I/O write (such asfor checkpointing). Or, a user might simply want to do a "gather-scatter" communication pattern as done in [6]. Either way, such patterns would benefit from having a means to combine multiple persistent requests into a single, manageable persistent request. With schedules, a user could easily define new collectives not defined by MPI as well as provide new implementations of existing collective operation that are defined by MPI.
  We recognize three options for API design here:
  1) assert that generalized requests, as currently defined, solves this need already (we discuss why this is not ideal in Subsection II-B below);
  2) allow the user to associate existing persistent requests together with each other somehow (e.g., with INFO to each initialization procedure warning MPI that this request will be cobbled together with others), plus a new set of mechanisms to achieve that connectivity;
  3) design and expose a new API that provides to the user the schedule creation and manipulation functionality that is currently internal-only inside MPI libraries (e.g., expose all the internal operations in Open MPI [5] (which derived from LibNBC [4]): ompi_sched_add_send, functions, renamed with MPI prefix, of course, and add new functions to manage the lifetime of MPIX_Schedule objects, like:
     - MPIX_Schedule_create(inout SCHED)
     - MPIX_Schedule_commit(in SCHED,
       out MPI_Request)
     - MPIX_Schedule_free(inout SCHED)

  In this paper, we employ a hybrid of the latter two approaches, while discarding the current formulation of generalized requests as a viable option.

The next subsections discuss other MPI technologies and further background and motivations for this work.

B. MPI Generalized Requests

MPI provides baseline support for generalized requests [7]. An MPI generalized request provides users with the ability to develop their own nonblocking functions for both point-to-point and collective scenarios. However, the MPI documentation states that the OS is responsible for concurrent execution, and thus MPI only supports ways of enabling concurrency mechanisms to interact with MPI. As such, generalized requests are defined by a set of user-provided functions that are called when certain MPI actions (i.e., MPI_WAIT, MPI_REQUEST_FREE, MPI_CANCEL, etc.) are performed on the generalized request. An example use-case of generalized requests in MPI is when an application has both a communication thread and a computation thread. While the application is responsible for making and managing these threads, the communication thread has the same means to interact with MPI as the computation thread would. The computation thread can call certain MPI functions to check on the progress of the request, but cannot be notified by MPI about any intermittent progress of the request. In addition to managing the communication, the communication thread would also be responsible for spending time progressing the generalized request and marking it as complete later, since MPI does not allow the user to hook their generalized request into the progress engine in any meaningful way. Furthermore, the additional thread progressing the generalized request is also responsible for performing the operations of the request (i.e., making the point-to-point, collective MPI calls, local operations, etc.). Thus, the drawback to generalized requests can be summed up with how the interface provided by MPI requires that the user make independent progress for all generalized requests and then inform MPI when each one is complete. Certain efforts to generalize generalized requests has also been done (e.g., [8]).

We note that, of the 110 the MPI applications studied by Laguna et al. in [9], none of the applications analyzed utilize MPI generalized requests.

C. LibNBC and Open MPI Schedules

Hoefler and Lumsdaine created LibNBC [4] with the goal of creating non-blocking collective functions that enabled a better overlap of communication and computation. To build a collective within LibNBC, a series of operations and rounds must be combined to form the new collective's schedule. The rounds are ordered sequentially and all operations in a round must be completed before the round can move on to the next. A collective implementation within the library calls API functions to add different operations to the current round, switch to the next round, and to commit the schedule before starting it. Collective operations created within LibNBC are done in a separate context of the communicator to avoid interfering with regular user communication in MPI. Progress inside a round can be asynchronous. However, in order to progress to the next round, one of LibNBC's testing functions must be called, limiting its ability to be truly asynchronous.

LibPNBC [10], [11], developed by Morgan et al., is an extension to LibNBC that provides support for persistent collective operations in MPI. Open MPI derives its scheduling mechanism from LibNBC, which it also incorporates and generalizes as a component of that MPI implementation [5]. Users cannot access LibNBC to create schedules; the LibNBC API is not published at the user-level by Open MPI; it is used internally.

The user-level schedules proposed in this paper are better than libNBC/libPNBC/OMPI-schedules because they are user-accessible, support run-once starting and ending operations, and work with persistent operations, rather than late-binding MPI operations. (In future work, we we will make them fully Turing complete.)

III. Design

Our goal in providing extensions and enhancements for scheduled operations is to support both on-loaded and off-loaded operations. If one can offload a large fraction of a schedule for an operation to switches, for instance, then terabit line rates (with commensurate message-rates) become possible for MPI. Removing the CPU from the critical path as much as possible is a strong goal moving forward, especially for multi-stage communication operations that can be planned with immutable schedules. This paper, without loss of generality, focuses on on-loaded schedules for communication operations (executed in software by the CPU).

In the remainder of this section, we cover the syntax and semantics of our proposed extensions to MPI for user-level scheduled communication.

A. Operation APIs - Syntax and Semantics

The design of the API is similar to that of LibNBC, and follows similar semantics with rounds and operations, at least at the user level (although, as an important distinction, our operations are always persistent MPI-4 operations[1]). Users can create a round, add as many operations to it as necessary, and then repeat the process to make as many rounds as needed. Once the user is satisfied with the schedule, they can commit the schedule to finalize it and get a schedule request (which is a persistent request). Once the user is ready to launch the operation, the user can start request using MPI_START (or MPI_STARTALL, if appropriate). One notable difference from LibNBC is that the sub-requests introduced in the schedule do not have to progressed by user code; instead, they are progressed by MPI's progress engine. Another distinction is how the requests are added to the schedule. This API requires users first to initialize all of the requests they wish to use in the schedule (as opposed to having specific API functions for adding each operation, like NBC_Sched_send). Through the use of functions like MPI_BCAST_INIT (presented in the MPI-4.0 standard), current persistent operations, like MPI_SEND_INIT, and partitioned communication functions, like MPIX_PARTITIONED_SEND_INIT from finepoints [3], users obtain a request handle to the specific operation, and then can pass that request handle to the schedule to insert it as a sub-request. The remainder of this section explores the functions that are proposed for the user-level API.

MPIX_SCHEDULE_CREATE(schedule, auto_free)

INOUT	schedule	schedule handle
IN	auto_free	a boolean indicating auto-freeing

[1]Relaxing this restriction is left for future work.

MPIX_SCHEDULE_CREATE is the first function that a user would use to build a schedule. This function is designed to create an empty schedule and return a handle so they can begin using rounds and operations to build their operations incrementally. The created schedule also has the option to automatically free its requests when the schedule itself is freed.

MPIX_SCHEDULE_ADD_OPERATION(schedule, request, auto_free)

INOUT	schedule	schedule handle
IN	request	request handle
IN	auto_free	a boolean indicating auto-freeing

Once the user has obtained a request for some operation that they wish to add to the schedule, the request can be passed to MPIX_SCHEDULE_ADD_OPERATION to be added as a sub-request to the specified schedule. Request handles returned from MPI_SEND_INIT and other communications operations are an example of operations that can be added this way.

Like the schedule itself, one can mark the requests as auto-freeing, which means that when the schedule request is freed, any sub-request that is marked auto free will also be freed. Any request that is not marked will allow the user to regain use of a valid, inactive handle after freeing the schedule. More specific semantics are discussed in the next section.

MPIX_SCHEDULE_ADD_MPI_OPERATION(schedule, mpi_op, invec, inoutvec, len, datatype)

INOUT	schedule	schedule handle
IN	mpi_op	the MPI_Op to perform
IN	invec	the first operand
INOUT	inoutvec	the second operand
IN	len	how many items to perform operation on
IN	datatype	the datatype of the operands

Additionally, a user may wish to perform some sort of operation on the data between communication steps. In fact, in order to properly build a reduction into a schedule, a user must have some way to perform a calculation on the data between steps. This function provides that capability by supporting the addition of any MPI_Op to the schedule. The user specifies the MPI_Op (such as MPI_MAX, MPI_SUM, etc.) and the location of the two operands. The results of the operation are stored at the location of inoutvec. User can also provide any user-created MPI_Op to the schedule[2]. In this case, the progress engine will call the user function described in the operation and pass the last four parameters to the user function.

MPIX_SCHEDULE_MARK_RESET_POINT(schedule)

INOUT	schedule	schedule handle

MPIX_SCHEDULE_MARK_RESET_POINT allows the user to specify that all rounds before the current round constitute rounds with operations that are only ever needed to be done the first time the schedule is executed[3]. Such rounds could be used to perform setup or synchronization before the main workload is to be performed. Every time the schedule is launched afterwards, the schedule will start from the round after the reset point. Calling this function several times while building a schedule will simply move the reset point to the current round. If a user never calls this function when creating their schedule, the reset point is by default the first round; all rounds will be performed every time the schedule is executed.

MPIX_SCHEDULE_MARK_COMPLETION_POINT(schedule)

INOUT	schedule	schedule handle

Completion points allow the user to specify that any further rounds will only be executed the final time a given schedule is executed. When a schedule reaches this point, it will be treated as the end of the schedule and be reset back to the reset point. The execute-once portion of the schedule that is after the completion point will only be executed by the progress engine once the request has been marked for freeing using the MPI_REQUEST_FREE function or during MPI_FINALIZE (whichever occurs first). There is currently no way in MPI to guarantee resource recovery before MPI_FINALIZE. However, it is good practice to call the relevant freeing function as soon as the object or handle is no longer needed, as it gives MPI the opportunity to execute clean-up actions earlier. Similar rules as MPIX_SCHEDULE_MARK_RESET_POINT apply when trying to mark multiple completion points. If the function is never used on a schedule, then the last round is assumed to be the completion point. One could use a round after the completion point to perform tear-down or synchronizing action.

MPIX_SCHEDULE_CREATE_ROUND(schedule)

INOUT	schedule	schedule handle

After adding several operations to the schedule, the user may wish to move onto the next round. To do so, the user would call MPIX_SCHEDULE_CREATE_ROUND. This function ends the current round by adding a new round to the schedule. Any operations that are added after this call are put into the new round. It is not valid to have an empty round in the schedule, so calling MPIX_SCHEDULE_CREATE_ROUND when the current round is empty will not create an additional round. Once the user is finished with the overall schedule, they do not need to call MPIX_SCHEDULE_CREATE_ROUND

[2]For the specific semantics of user-defined operations, see Section 5.9.5 in the MPI 3.1 standard [7].

[3]This concept does not exist in LibNBC/Open MPI schedules.

an additional time to “finalize” the last round. They should instead call MPIX_SCHEDULE_COMMIT.

MPIX_SCHEDULE_COMMIT(schedule, request)

IN	schedule	schedule handle
OUT	request	request handle

MPIX_SCHEDULE_COMMIT should be called when the user has completed the building phase. When the user calls this function, they receive a handle to a request that can be passed to MPI_START to launch the schedule. Additionally, this request can be passed to MPI completing functions, such as MPI_WAIT and MPI_TEST. A user could also potentially nest schedules together to create a hierarchy of operations by passing this reuqest into another schedule. Should a schedule be committed with the current round lacking any operations to perform, the empty round will be trimmed from the schedule since there is nothing for the progress engine to do. If commit is called on a schedule with only one round and no operations, an error will be returned and the request handle will not be valid. After calling MPIX_SCHEDULE_COMMIT, it is not valid to add more operations, rounds, reset points, or completion points to the schedule.

MPIX_SCHEDULE_FREE(schedule)

INOUT	schedule	schedule handle

This function is used to free the schedule specified. After a schedule has been freed, it is not valid to perform any more actions on the schedule. Any corresponding request created using the freed schedule can be also freed with MPI_REQUEST_FREE. Additional considerations for freeing the requests will be discussed in the next section.

B. Operational Semantics

The prior section presents an overview of the API functions we propose, along with description of these operations and constraints on their use. This section discusses additional considerations about the properties of the proposed API that affect its operational semantics:

1) Once a request is bound to a schedule (and thereby used in a new composite communication operation), it can no longer be used outside of that schedule. Only one schedule can own a request at a time; it is not valid to add the same request to two different schedules. However, because our API requires the user to provide every request they wish to add to the schedule, the user will have access to the request handles after adding them to the schedule. As these requests are normal MPI_Requests, allowing the users to call some MPI functions on them could prove to be useful. It is reasonable that a user might find it useful to check on the progress of a specific sub-request inside the schedule. Thus, we have required that operations such as MPI_WAIT and MPI_TEST remain valid on requests that are a part of the schedule. This allows a user to retrain fine-grained control over the application, by allowing processes to react to the completion of specific parts of the schedule. However, calling MPI functions that could result in the individual requests being modified, such as MPI_REQUEST_FREE and MPI_START, is not permitted. Calling these functions on requests currently within a schedule could break the concurrency provided by that schedule or produce undefined behavior on the resources associated with the requests.
2) The steps of creating, building, and committing a schedule together constitute the initialization stage of a persistent MPI operation. The resulting MPI_Request handle represents that persistent operation, which can be used to start the operation (e.g., using MPI_START) to complete the operation (e.g., using either MPI_TEST or MPI_WAIT), and to free the operation (i.e., using MPI_REQUEST_FREE). The rules for starting a persistent operation created by committing a schedule depend on the rules for starting the constituent operations from which it was built. If any of the constituent operations requires a particular start order, then the composite operation must also adhere to that restriction. For example, if a persistent point-to-point operation is added to a schedule, then the resulting composite operation must be started at a time (and in an order relative to other MPI function calls) that permits correct matching of the point-to-point operation with the corresponding operation at the destination (or source) MPI process. Similarly, if a persistent collective operation is added to a schedule, then the resulting composite operation must be started at a time (and in an order relative to other MPI function calls) that complies with the value given for the mpi_assert_strict_start_order MPI_Info key.
If the mpi_assert_strict_start_order assertion is not used with a given persistent collective operation (that is, defined on the communicator from which the operation was created), then MPI has complete freedom of ordering with regard to all other operations MPI does with respect to each process involved. If that assertion is used, however, then the scheduled operation must be started across the group of each communicator associated with it in a way to keep the start order consistent across the whole group.
3) When building the schedule and the request objects, it makes sense that, at some point, these objects will need to be freed. As touched on above, both the schedule and the request have a parameter that allow the user to specify whether or not they will be freed when a function is called. By default, all requests will be automatically freed when the schedule request is freed. As such, the sub-requests will not be viable after the conglomerate

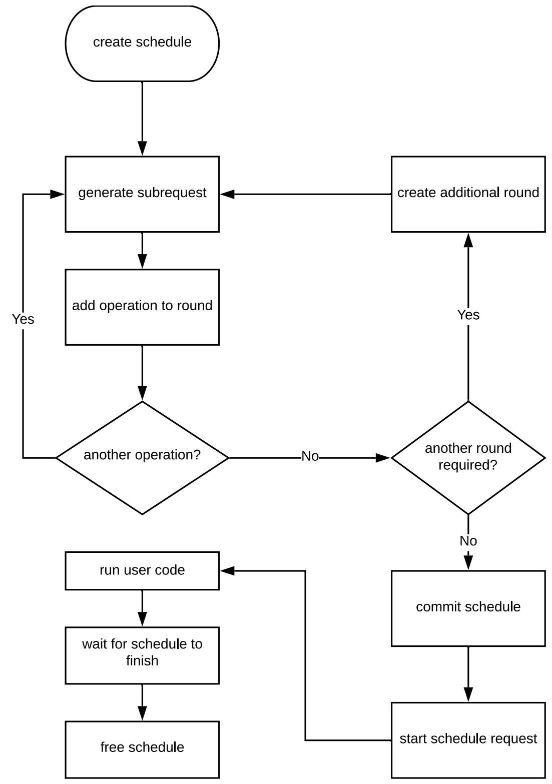


Fig. 1. Procedure of schedule creation

operation is freed[4].

C. Examples

An example flow for building a schedule is shown in Fig. 1. Using this flow, sample pseudo-code is presented in Fig. 2. This code shows how to combine the new API with the existing concept of adding sends, receives, and collectives to the schedule from LibNBC. Roughly, this code is building a schedule for a reduction to non-zero rank from a reduction to zero-rank and a point-to-point operation. All parameters not directly related to building the schedule are elidedfor brevity. The use of auto-freeing has been omitted for brevity.

IV. Qualitatively Modeling Performance

In this section, we consider qualitative modeling concepts to show the value of user-level scheduled requests. If those scheduled requests are offloaded to a progress engine that works asynchronously from the user thread, a considerable amount of time can be saved, especially if the schedule is persistent and used often. Fig. 3 shows how this is possible by demonstrating the timing of one application cycle. Note that after the first cycle, all applications will start from the same spot, time Tn.

[4]Freeing described is in respect to how the MPI Standard talks about MPI managing resources from the user. While some languages have the ability to handle allocation and de-allocation of resources for the user, MPI itself generally does not. Thus, to maintain portability, we provide these freeing options

```
//Create & initialize schedule
MPIX_Schedule sched;
MPIX_Schedule_init(&sched);

//Add reduce operation to schedule
MPI_Request reduce_op;
MPI_Reduce_init(<reduce parameters,
root = 0>, &reduce_op);
MPIX_Schedule_add_operation(&sched,
reduce_op);

//Prior ops only happen on first run
MPIX_Schedule_mark_reset_point(&sched);

MPIX_Schedule_create_round(&sched);

//Add more operations
if (0 == my_rank) {
    MPI_Request ssr;
    MPI_Send_init(<send parameters w/
    dest = root>, &ssr);
    MPIX_Schedule_add_operation(&sched,
    &ssr);
}
else if (root_rank == my_rank) {
    MPI_Request rsr;
    MPI_Recv_init(<recv parameters -
    source = 0>, &rsr);
    MPIX_Schedule_add_operation(&sched,
    &rsr);
}
//Create request handle for schedule
MPI_Request request;
MPIX_Schedule_commit(&sched, &request);

//Start schedule
MPI_Start(request);

//   Other user code to run   //
// while schedule is executed //

//Wait for schedule to complete
MPI_Wait(request, &status);
MPI_Request_free(&request);
MPIX_Schedule_free(&sched);
```

Fig. 2. Scheduled communication example

In this example, a user application has six equally sized workloads to do, all of which can be done independently from any communication needed to be done. At the same time, the user application also needs to do three communication sequences. While not specifically intertwined, these nine operations must be completed before the user application can

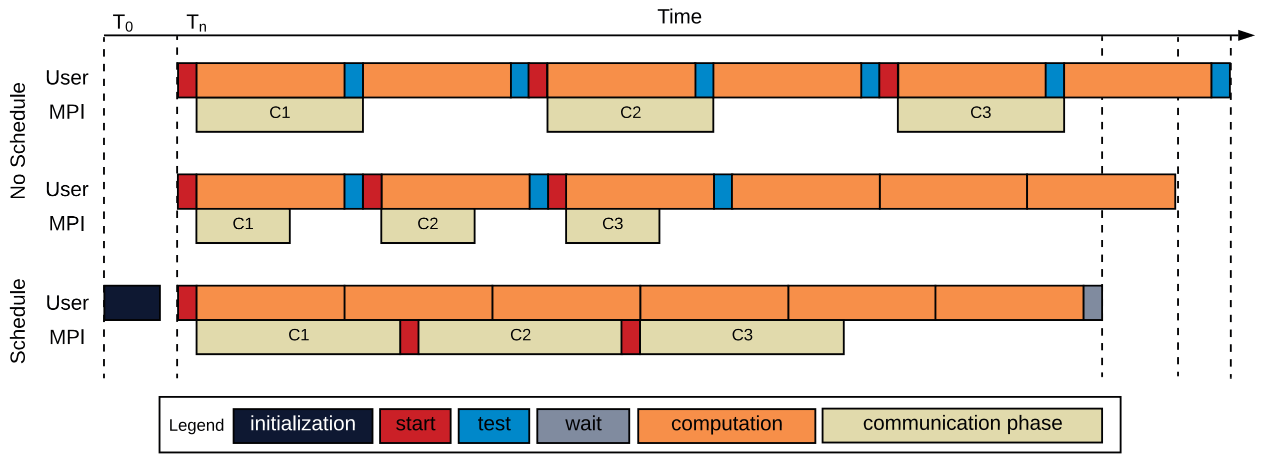


Fig. 3. A comparison between a user manually progressing communication and a user using a schedule (bottom). All computational blocks are of equal work. Both the worst-case for manual progression, where communication ends just after the user checks (top), and the best-case, where communication ends before the user checks (bottom), are presented. As shown, the use of a user-level schedule allows for a reduced time-to-solution due to the removal of test operations. A one-time cost of building the schedule is now required, but is inexpensive if the schedule is used multiple times.

proceed to the next step of the code, and these nine operations will be repeated many times during the user's application. In current MPI code, an application wishing to start this process would first start the first communication sequence and then begin computations. In the first example in Fig. 3, the user's application completes the first computation sequence just before the first communication sequence finishes. When the user tests to see if the communication sequence has completed, they see that it has not and proceed to start another computation sequence. After completing those computations, the user application tests again, finds that the communication sequence has finished, and starts the next round of communication. Unfortunately for the user, the communication sequences continue to run long, and the application achieves the worst case scenario of communication and computation overlap.

The second example in the figure represents the opposite scenario; now the user application has achieved good overlap between communication and computation as a result of communication times being quicker. Here, every time the user is ready to test for completion of a communication sequence, they find that it has completed and can immediately progress communication to the next sequence. Additionally the user is also able to finish all necessary communications after its third computation sequence, and thus does not have to spend any extra time by testing and launching more communications later on (at least in the context of this portion of the code). This example shows how achieving good communication and computation overlap can help the program achieve a shorter time to completion. However, we can do even better with user-level scheduled requests.

Consider the third example in the diagram. Here the user application is using an MPI implementation that supports user-level schedules. In this code, the user absorbs the one-time cost spent on creating the persistent communication sequences (building the schedule, time T0 to Tn) but, does not have to spend any time subsequently testing for communication completion. Once created, the user must also offload this communication sequence to the progress engine. Afterwards, the user is free to do all the computation sequences, and does not need to waste time progressing the communication sequences, as the progress engine uses the provided schedule to do so. When the application is ready to check on the progress, it can wait to see if the schedule is complete. Since the schedule is persistent, the user can use the schedule right away on subsequent uses; there is no cost to restart the schedule, only the initialization and building cost on the first use.

The above examples assume that the computation takes longer than the communication. Next, we illustrate that even with longer communication time than computation time, better overlap is still achieved by using the schedule alongside a strong progress engine. Fig. 4 depicts this scenario. In this situation, both scenarios end up completing all possible computation they can, and then must wait on the communication sequences to complete. However, when the application is directly managing all the constituent communication operations, it must spend time manually progressing multiple requests (through repeated calls into MPI completing procedures, like MPI_TEST). For the program using a schedule, the progress engine takes care of moving all the constituent communication sequences along.

V. Implementation

In this section, we first describe the underlying MPI implementation, ExaMPI. We focus on its key features, as well as its strong progress engine and support for schedules. With that understanding, we then discuss the implementation of schedules in ExaMPI.

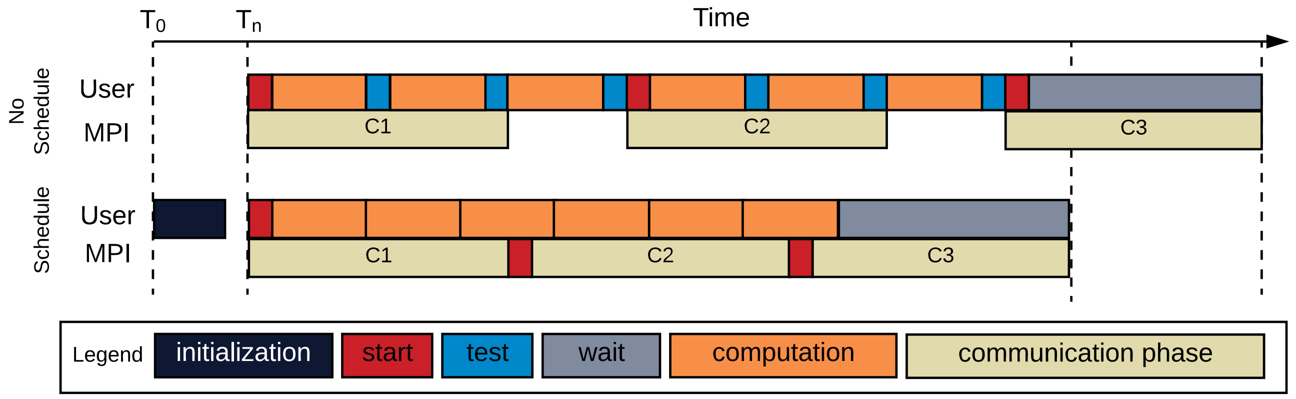


Fig. 4. Another comparison between a user manually progressing communication (top) and a user using a schedule (bottom). All computational and communication blocks are of equal size. As shown, offloaded schedules allow for reduced time-to-solution due to the removal of test operations by the user and the ability of the user to leverage the progress engine to progress communication. The overhead of initialization is again amortized across repetitions of the offloaded schedule.

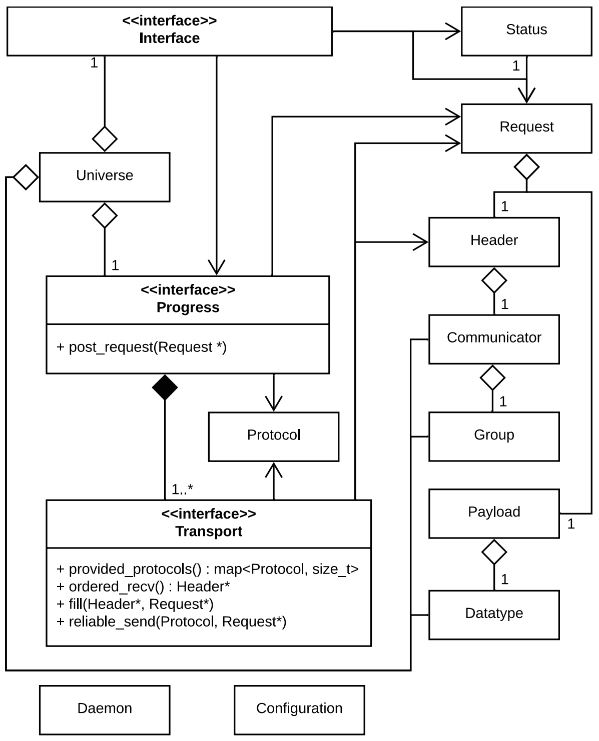


Fig. 5. Design overview of ExaMPI's structure (adopted from [1])

A. ExaMPI

ExaMPI [1] is a C++17-based library designed for modularity, extensibility, and understandability. ExaMPI's high-level structure is depicted in Fig. 5. The design ensures internal interfaces through C++ classes, which allows replacement of behavior with ease. The code base supports both native C++ threading with thread-safe data structures and a modular progress engine. In addition, the transport abstraction implements UDP, TCP, OFED verbs, and LibFabrics for high-performance networks.

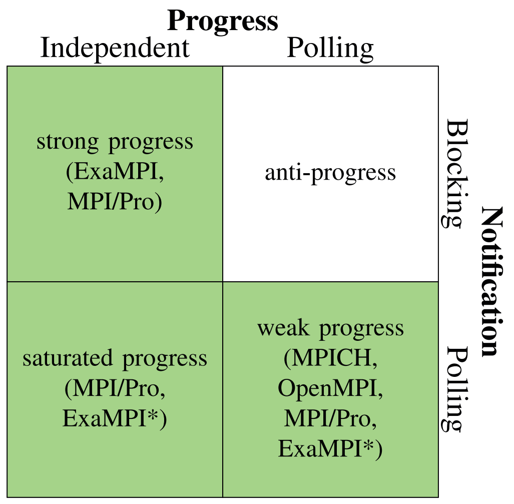

Progress
Independent
Polling
Notification
Blocking
Polling
strong progress (ExaMPI, MPI/Pro)
anti-progress
saturated progress (MPI/Pro, ExaMPI*)
weak progress (MPICH, OpenMPI, MPI/Pro, ExaMPI*)

Fig. 6. Dimitrov's Progress and Notification Classification Diagram (adopted from [12], [1]); *Forthcoming modes in ExaMPI.

The progress engine interface shown in Fig. 5 specifies a minimal set of functions required. Through this interface, all modes of progress and completion are supported as depicted in Fig. 6. The strong progress engine present in ExaMPI allows a variable number of progress threads and supports more modularity internally to support different matching algorithms and decision functions about operation algorithm usage. The overall design allows for offloading of any operations or even the entire progress engine onto hardware.

Despite being an immature implementation that is not perfectly tuned yet, we still chose to implement user-level schedules in ExaMPI due its support for a strong progress engine. Without a strong progress engine, we cannot experiment with the offloading potential of the schedules and achieve the benefits outlined in Fig. 3 and Fig. 4.

B. Building New Operations

As previously discussed, a schedule consists of several rounds, each containing at least one operation. A round contains one or more sub-requests that can be launched concurrently and completed in any order. Sequencing operations requires multiple rounds. Within ExaMPI, these sub-requests are represented by specialized internal requests that enable them to have a reference back to their round, as well as give them their own set of methods that allow them to behave appropriately. The rounds themselves have no reference for where they reside in the schedule; they only know if they have a next round to launch when their time comes. If the round does not have a subsequent round to launch, the progress engine assumes that it is the last round, and marks the schedule request as complete. This assumption comes from the basis that it is not possible to build a schedule with multiple final rounds. As per the flow diagram in Fig. 1, in order to make a new round, the rounds must be first linked. Only the last round is allowed to not have a link to another round. The design (and user-level API) forbids a round from having multiple next rounds, which in turn prevents multiple final rounds. Ideally, the user-built schedule would behave logically as if the user called the normal MPI functions in the same order as defined in the schedule.

C. Progressing Operations

Internally, the sub-requests added to the rounds are a special type of request that behave differently when the progress engine finishes with them. To the engine itself, it is not aware that there is such a distinction. To make the sub-requests unique, they override the function that is called when the progress engine wants to release a request it has finished. Instead of only marking itself as complete and notifying anyone waiting on this request, these special requests also interact with their respective round. When a sub-request is completed, it tells its round that it is done, which increments a completion counter inside the round. To avoid a race condition from a multi-threaded progress engine, only one sub-request may access the round's counter at a time. After successfully incrementing the counter, the sub-request checks with the round to see if it is the last one to arrive. If it was, the sub-request then tells the round to progress to the next round. The current round will then take care of launching the next round, which means launching its batch of sub-requests[5]. But, if the round does not have a next round, it assumes that it is the final round (as mentioned above) and that it is the round responsible to marking the original schedule request as completed. Fig. 9 shows the aforementioned flow from the progress engine's side.

[5]While operations in a round should not have a specific ordering, the launching of a round will likely result in a first-in, first-out style of starting for the sub-requests. Additionally, the completion order will be purely up to how to the progress engine progresses the requests.

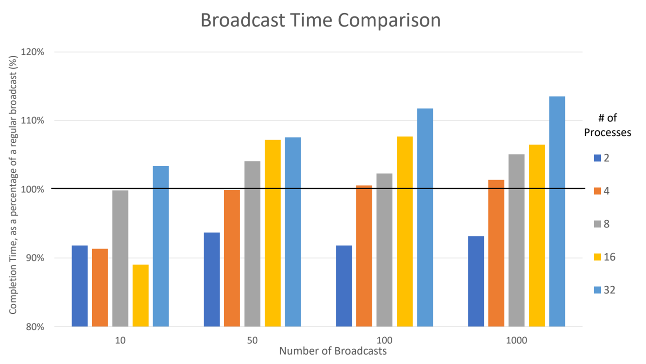


Fig. 7. A comparison of regular broadcast times versus schedule broadcast times. The black line represents the time of a regular broadcast. Any times over the line are slower than the regular broadcast and any times under the line are faster.

VI. Results

We have evaluated our implementation using ExaMPI [1]. We first measured the overhead of schedule creation and our measurements indicate that the schedule creation overhead is low (less than 1 ms, on average). Fig. 7 shows time required to complete the standard blocking broadcast compared to our broadcast with schedule. We ran multiple tests with varying number of broadcast (10, 50, 100, 1000) to understand the benefits of using a schedule multiple times and with a varying number of processes (2, 4, 8, 16, 32) to gain some insight into scalability. In this figure, a bar of 150% means that using a scheduled broadcast in this scenario, on average, was 50% slower than a regular MPI broadcast; a bar of 50% means that using a scheduled broadcast is 50% faster than a regular MPI broadcast.

From the figure, we see that these early results indicate that as one uses the schedule more often, there is not an insignificant increase in delay, which means that the user is not increasing time to completion when using a schedule. While increasing the number of processes does appear to increase the time of executing a scheduled broadcast, this slight extra time is likely the result of the small overheads of the schedule itself. When comparing to normal MPI collective functions, we expect the schedules to have a slight overhead from managing the starting of rounds. However, the strongest feature of these schedules is their ability to create unique communication patterns that are not in the MPI implementation. With the results shown in this figure, it indicates that schedules can be effective for this type of communication too.

With ExaMPI's strong progress engine, we were successfully able to offload the schedule to the progress engine and achieve a 98%+ gap to do computations, as shown in Fig. 8 This is important because it means that as we starting scaling schedules to even more processes, we have that much more time to overlap computation with the communications done in the schedule.

Interestingly, it appears that in the case of two processes (where a broadcast is only one send and one receive), the

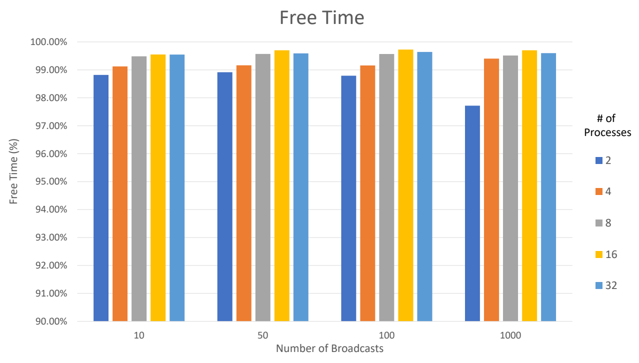


Fig. 8. Average free time when using schedule (higher is better)

scheduled broadcast was faster, perhaps indicating an extra overhead in ExaMPI; we will revisit that performance differential by further study of ExaMPI.

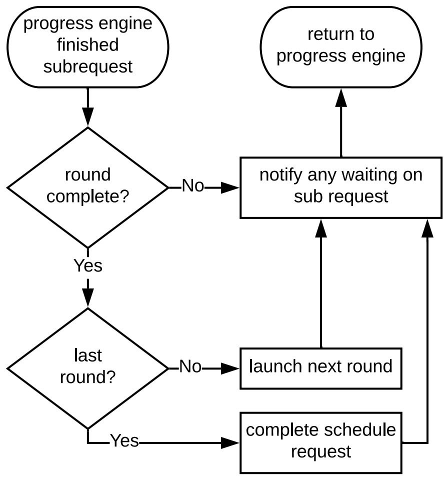
VII. Extensions and Future Work

MPI supports no asynchronous notification methodology, such as the event-driven model shown in PERUSE [13] and MPI/RT (a priority, event model) [14] efficiently. While PERUSE was designed for event-driven notification of MPI implementation information and worked successfully with certain MPIs such as MPI/Pro [15] and Open MPI, it was never adopted by the Forum. Instead, the Forum has encouraged polling-base notification of events for MPI-4 tools [16], [17] (slides 18–27). MPI could well be elaborated to support event-driven programming, using the thread-levels described in the MPI-4 tools proposal for callbacks (with extensions to allow hardware offload through downcalls vs. upcalls). Such callbacks would allow events generated by MPI applications to be local, groupwise, or external in origin. For instance, this model allows layering on top of MPI of certain operations, such as MPIX_COMM_REVOKE[6] in ULFM [18], to be implemented as a groupwise event that calls the error handler on a communicator across a group, rather than as a monolithic new API to be standardized. Another example is the MPIX_PREADY API proposed with partitioned communication [3], [17] (slides 38–45); it constitutes an event being raised by an MPI application to tell MPI that a message partition is complete and available to MPI for transmission.

The concepts presented here can be extended, with care, to MPI one-sided communication schedules as well, but require non-blocking constructors and destructors for windows and persistent versions of operations such as MPI_WIN_FENCE. Some of us and others are studying such extensions to MPI.

A further opportunity for schedules defined using this API is the potential for offloading them completely to FPGAs. Since schedules are persistent, their MPIX_SCHEDULE_COMMIT operation could include programmatic transfer to an FPGA,

[6]Despite its mention here, this paper does not endorse MPIX_COMM_REVOKE as a good approach to error propagation in a fault-tolerant MPI.



progress engine finished subrequest
round complete?
No
Yes
last round?
No
Yes
return to progress engine
notify any waiting on sub request
launch next round
complete schedule request

Fig. 9. Progress engine progressing a request

with the potential for either using a state machine on this device (or potentially compiling and loading new FPGA logic through partial reprogramming for such as schedule).

Finally, there is a desire to eventually make user-level schedules Turing complete. While the authors acknowledge that the design presented in this paper represent a DAG more than a Turing complete implementation, the user-level schedules are only a few steps from Turing complete. In the future, we plan to implement the ability to repeat operations in a schedule for a user-specified number of times and provide conditionals to decide whether a certain operation will be performed or not.

VIII. Conclusion

The introduction of persistent collective communication and forthcoming partitioned point-to-point communication in MPI-4.0 (which provides full channels for MPI) has opened the opportunity to introduce another important extension to MPI; namely, user-level scheduled communication. Generalized requests, introduced in MPI-2, do not meet the needs for effective generalized communication extensions.

This paper introduces user-level scheduled communication, an extensible interface motivated in part by internal concurrent schedule interfaces in certain MPI libraries. Background, motivation, design, implementation and qualitative performance modeling were described. Implementation with the ExaMPI research MPI implementation was highlighted. The value of strong progress in supporting these operations was mentioned.

The schedules described here provide DAG-type operations including the ability to do multiple rounds of communication, and to include, hierarchically, both point-to-point and collective operations inside schedules. This differs from and extends the schedules implemented in libraries such as Open MPI, which are late-binding, point-to-point based and not part of the user API. Benefits of this extension to MPI include the ability for high-performance collective operations to be added without working within an MPI library itself, as well as introducing new communication patterns at high performance. These operations would also be cross-MPI portable if user-level schedules are standardized by the MPI Forum in future.

Early performance results indicate that schedule formation is not overly expensive, considering that schedules will normally be used many times once created. Also, performance indications show that schedules work well with ExaMPI's strong progress engine, providing the potential for overlap of communication and computation while schedules are active.

While this paper confines itself to persistent message passing, concepts described here generalize to support schedules of non-persistent MPI operations, as well as one-sided communication operation. One-sided operations could be scheduled provided certain additional operations are added to the standard (e.g., fully non-blocking window creation). Further, if combined with event-based extensions to MPI (asynchronous notification), these operations provide a basis for full task-based parallel programming with message passing and will help enhance scheduling and triggering of MPI+X operation sequences (e.g., MPI+CUDA).

Acknowledgment

This work was supported in part by the National Science Foundation (NSF) under Grants Nos. CCF-1925603, CCF-1822191, and CCF-1821431. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. This work was part-funded by the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement 801039 (the EPiGRAM-HS project).

The authors wish to thank Dr. Puri Bangalore for his helpful feedback on this paper.

References

[1] A. Skjellum, M. Rüfenacht, N. Sultana, D. Schafer, I. Laguna, and K. Mohror, "ExaMPI: A new middleware architecture and implementation to accelerate innovation with the Message Passing Interface," Sept. 2019, proceedings of CARLA2019.

[2] P. V. Bangalore, R. Rabenseifner, D. J. Holmes, J. Jaeger, G. Mercier, C. Blaas-Schenner, and A. Skjellum, "Exposition, clarification, and expansion of MPI semantic terms and conventions: Is a nonblocking MPI function permitted to block?" 2019, accepted to EuroMPI 2019, in press.

[3] R. E. Grant, M. G. F. Dosanjh, M. J. Levenhagen, R. Brightwell, and A. Skjellum, "Finepoints: Partitioned multithreaded MPI communication," in *High Performance Computing - 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16-20, 2019, Proceedings*, ser. Lecture Notes in Computer Science, M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, Eds., vol. 11501. Springer, 2019, pp. 330–350. [Online]. Available: https://doi.org/10.1007/978-3-030-20656-7_17

[4] T. Hoefler and A. Lumsdaine, "Design, Implementation, and Usage of LibNBC," Open Systems Lab, Indiana University, Tech. Rep., Aug. 2006.

[5] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[6] J. Gong, S. Markidis, E. Laure, M. Otten, P. Fischer, and M. Min, "Nekbone performance on gpus with openacc and cuda fortran implementations," *The Journal of Supercomputing*, vol. 72, no. 11, pp. 4160–4180, Nov 2016. [Online]. Available: https://doi.org/10.1007/s11227-016-1744-5

[7] MPI Forum, "MPI: A message-passing interface standard. version 3.1," University of Tennessee, Knoxville, TN, USA, Tech. Rep., 2015.

[8] R. Latham, W. Gropp, R. B. Ross, and R. Thakur, "Extending the MPI-2 generalized request interface," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007, Proceedings*, ser. Lecture Notes in Computer Science, F. Cappello, T. Hérault, and J. J. Dongarra, Eds., vol. 4757. Springer, 2007, pp. 223–232. [Online]. Available: https://doi.org/10.1007/978-3-540-75416-9_33

[9] I. Laguna, K. Mohror, N. Sultana, M. Rüfenacht, R. Marshall, and A. Skjellum, "A large-scale study of MPI usage in open-source hpcapplications," in *Proc. of SC 2019*, November 2019, accepted, in press.

[10] B. Morgan, D. J. Holmes, A. Skjellum, P. Bangalore, and S. Sridharan, "Planning for performance: persistent collective operations for MPI," in *Proceedings of the 24th European MPI Users' Group Meeting, EuroMPI/USA 2017, Chicago, IL, USA, September 25-28, 2017*, A. J. Peña, P. Balaji, W. Gropp, and R. Thakur, Eds. ACM, 2017, pp. 4:1–4:11. [Online]. Available: https://doi.org/10.1145/3127024.3127028

[11] D. J. Holmes, B. Morgan, A. Skjellum, P. V. Bangalore, and S. Sridharan, "Planning for performance: Enhancing achievable performance for mpi through persistent collective operations," *Parallel Computing*, vol. 81, pp. 32 – 57, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819118302412

[12] R. P. Dimitrov, "Overlapping of communication and computation and early binding: Fundamental mechanisms for improving parallel performance on clusters of workstations," Ph.D. dissertation, Mississippi State University, Mississippi State, MS, USA, 2001.

[13] PERUSE Working Group, "MPI PERUSE an MPI extension for revealing unexposed implementation information, version 2.0," last downloaded: June 25, 2019. [Online]. Available: http://mpi–peruse.org/current_peruse_spec.pdf

[14] A. Skjellum, A. Kanevsky, Y. S. Dandass, J. Watts, S. Paavola, D. Cottel, G. Henley, L. S. Hebert, Z. Cui, and A. Rounbehler, "The real-time message passing interface standard (MPI/RT-1.1)," *Concurrency - Practice and Experience*, vol. 16, no. S1, pp. 0–322, 2004. [Online]. Available: https://doi.org/10.1002/cpe.744

[15] R. Dimitrov and A. Skjellum, "Software architecture and performance comparison of mpi/pro and MPICH," in *Computational Science - ICCS 2003, International Conference, Melbourne, Australia and St. Petersburg, Russia, June 2-4, 2003. Proceedings, Part III*, 2003, pp. 307–315.

[16] M.-A. Hermanns *et al.*, "Callback-driven event interface for MPI_T," June 2019, last downloaded, June 27, 2019. [Online]. Available: https://github.com/mpi-forum/mpi-issues/issues/79

[17] M. Schulz, D. J. Holmes, M.-A. Hermanns, and A. Skjellum, "The Message Passing Interface: Towards MPI 4.0 & beyond," June 2019, iSC 2019 MPI Forum BOF.

[18] W. Bland, A. Bouteiller, T. Hérault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An evaluation of user-level failure mitigation support in MPI," *Computing*, vol. 95, no. 12, pp. 1171–1184, 2013. [Online]. Available: https://doi.org/10.1007/s00607-013-0331-3

[19] A. Skjellum, N. E. Doss, K. Viswanathan, A. Chowdappa, and P. V. Bangalore, "Extending the message passing interface (MPI)," in *Proceedings Scalable Parallel Libraries Conference*. IEEE, 1994, pp. 106–118.

[20] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir, "MPI-2: extending the Message-Passing Interface," in *Euro-Par '96 parallel processing: second International Euro-Par Conference, Lyon, France, August 26–29, 1996: proceedings*, ser. Lecture notes in computer science, L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, Eds., vol. 1123–1124. Berlin, Germany / Heidelberg, Germany / London, UK / etc.: Springer-Verlag, 1996, pp. 128–135.

[21] A. Skjellum, "High performance MPI: Extending the Message Passing Interface for higher performance and higher predictability," in *International conference, Parallel and distributed processing techniques and applications*, H. Arabnia, Ed., 1998, pp. 25–32.