Mixing Ranks, Tasks, Progress and Nonblocking Collectives

Jean-Baptiste Besnard
jbbesnard@paratools.fr
ParaTools SAS
Bruyères-le-Châtel, France

Julien Jaeger
julien.jaeger@cea.fr
CEA, DAM, DIF
F-91297 Arpajon, France

Allen D. Malony
malony@paratools.com
ParaTools Inc.
Eugene, USA

Sameer Shende
sameer@paratools.com
ParaTools Inc.
Eugene, USA

Hugo Taboada
hugo.taboada@cea.fr
CEA, DAM, DIF
F-91297 Arpajon, France

Marc Pérache*
marc.perache@cea.fr
CEA, DAM, DIF
F-91297 Arpajon, France

Patrick Carribault
patrick.carribault@cea.fr
CEA, DAM, DIF
F-91297 Arpajon, France

ABSTRACT

Since the beginning, MPI has defined the rank as an implicit attribute associated with the MPI process' environment. In particular, each MPI process generally runs inside a given UNIX process and is associated with a fixed identifier in its `WORLD` communicator. However, this state of things is about to change with the rise of new abstractions such as MPI Sessions. In this paper, we propose to outline how such evolution could enable optimizations which were previously linked to specific MPI runtimes executing MPI processes in shared memory (e.g. thread-based MPI). By implementing runtime-level work-sharing through what we define as *MPI tasks*, enabling the ability to progress indifferently from stream context we show that there is potential for improved asynchronous progress. In the absence of a Session implementation, this assumption is validated in the context of a thread-based MPI where nonblocking Collective (NBC) were implemented on top of *Extended Generic Requests* progressed by any rank on the node thanks to an MPI extension enabling threads to dynamically share their MPI context.

CCS CONCEPTS

• **Computing methodologies → Parallel programming languages**; *Shared memory algorithms.*

KEYWORDS

Hybrid MPI, Thread-based MPI, MPI Sessions, nonblocking collectives, Progress

*Also with CReSTIC Laboratory EA3804, University of Reims Champagne-Ardenne.

ACM Reference Format:
Jean-Baptiste Besnard, Julien Jaeger, Allen D. Malony, Sameer Shende, Hugo Taboada, Marc Pérache, and Patrick Carribault. 2019. Mixing Ranks, Tasks, Progress and Nonblocking Collectives. In *26th European MPI Users' Group Meeting (EuroMPI 2019), September 11–13, 2019, Zürich, Switzerland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3343211.3343221

1 INTRODUCTION

Since it first appeared in 1994, the Message Passing Interface (MPI) API has become the *de facto* standard for distributed memory programming. MPI is most commonly used to run SPMD parallel programs on HPC platforms and is highly optimized for relatively static communicators (e.g., `MPI_COMM_WORLD`) and two-sided communications to achieve the highest performance possible over HPC hardware. MPI allows programs to run in other configurations, including multiple codes sharing the same communication space (MPMD) or creating dynamic processes (Spawn, Connect). In such cases, the user is responsible for managing process interactions, using MPI rank identifiers to specialize operations and data sharing.

In this paper, we question the role of MPI context in current and future standards. We argue that MPI's implicit distributed memory model limits its integration in highly concurrent programming environments. With the rapid increase of the number of cores on a single node, applications are relying more on hybridization, combining MPI internode parallelism with a shared-memory model. This puts pressure on MPI's interfaces and runtime system to evolve new means of managing parallelism. The paper first surveys MPI hybridization progress. Our evaluation leads us to the conclusion that a new parallel abstraction within MPI is necessary. Specifically, we propose the concept of *MPI Tasks* as execution-stream independent MPI operations, easing the expression of fine-grain parallelism involving communications.

The second part of the paper explores the practicalities of implementing communicating tasks in MPI. In particular, we focus on a partially standardized feature called *Generalized Requests* [17] and their non-standard derivative *Extended Generalized Requests* (EGREQ) [26]. We discuss how they approximate a *task* in the MPI standard and then describe their implementation in the MPC thread-based MPI runtime [28]. In particular, we consider a *disguise* capability enabling MPI processes to temporarily borrow another rank's MPI context. Such support allowed us to implement *MPI Tasks* inside our runtime as independently scheduled MPI communications, potentially crossing standard MPI context boundaries. To demonstrate these capabilities, we describe our proof-of-concept implementation of nonblocking collectives (NBCs) [21] using *MPI Tasks*, comparing it with our previous integration of libNBC [8].

The paper ends with a general discussion on the promise of MPI Tasks combined with upcoming MPI Sessions [22] in the standard. In particular, we advocate for potential means of communicating between Sessions with a more relaxed context, for instance, using Remote Procedure Calls (RPCs) or Active Messages (AMs) to realize plausible coupling scenarios.

2 RUNTIME WORK-SHARING OVERVIEW

The idea of runtime work-sharing with respect to MPI applies to all the operations done by the MPI runtime at the node level. For instance, communication operations include matching [14], data copies, reductions, data-progress protocols, and so on. Processing all these operations in parallel is beneficial, both in terms of throughput and asynchronous activity. Indeed, communications are often an important sequential contributor to limiting speedup, while asynchronous communications can overlap processing steps thereby mitigating their cost with respect to the calling execution stream. If runtime work-sharing can be enhanced, it offers an advantage of requiring minimal to no effort from the end-user to provide improved hybridization.

2.1 Process-Based MPI Runtimes

Let us consider how "regular" production-grade MPI runtimes operate. Indeed, in their default configuration, runtimes do not progress communications outside of MPI. We witnessed this in prior work characterizing the embedding of MPI nonblocking communications inside OpenMP tasks [6], requiring manual progress requests (e.g., with `MPI_Test`). nonblocking does not mean asynchronous and this is a perfectly valid interpretation of the standard. Nonetheless, from an end-user perspective, the lack of overlapping communication with the computation is surprising. A reason for this is latency oriented optimization which tends at minimizing extra synchronization imposed by a progress thread with potential added noise and cost [33].

From a serial MPI process stance, progressing MPI solely during MPI calls is then a reasonable model, avoiding locks and yielding the best message rate. However, this can be counterproductive in times where a single MPI process can host hundreds of execution streams, posing challenges that following models try to address.

2.2 Thread-Based and co-Located MPI Runtimes

Some runtimes explored configurations where MPI "processes" are less isolated in comparison to running in regular UNIX processes. Several implementations have been considered, with the core idea of leveraging a shared address space for inter-operation[15, 23–25, 28, 31]. The immediate advantage of these models is that an MPI process can easily share their context, for example, copies and message matching. In addition, the network context can be shared between multiple ranks, avoiding memory replication. Eventually, this enables higher message throughput, as messages are simple *memcpy* in shared memory. However, this comes with its constraints [27]. For instance, in MPC, global process variables must be managed at compilation time or by process-image replication [4]. In addition, such runtimes are by nature thread-multiple and usually yield lower inter-node message rates due to extra locking required [33].

2.3 MPI Sessions

MPI Sessions [22] which are in the process of being standardized [13] are a mean of launching MPI in a more scalable manner. In particular, *MPI_Init* is replaced with a group predating the existence of the main communicator. Despite being fully transposable to "legacy" MPI codes, this has some powerful implications for MPI runtimes. In particular, multiple instances of the runtime may be collocated in the same UNIX process, being instantiated by distinct libraries. Despite not communicating with each other, runtime instances could collaborate by sharing their progress. Indeed, sessions do not define which execution stream is running a given MPI call, but they could be used to spawn serial sessions which could then be seen as thread-multiple interactions with an abstracted common MPI runtime. Thus, serial communication could participate in the progress of another session's operation mimicking what can be done with thread-based MPI at the runtime instance, instead of the rank level – point that we intend to explore later on in this paper.

2.4 Hardware MPI

Runtime work-sharing can benefit greatly from hardware support, specifically networking hardware with embedded MPI processing capabilities, such as MPI message matching as seen in the Bull BXI [2, 10], the Cray slingshot interconnect, or Mellanox hardware. The underlying interconnect is being enhanced with computing cores to handle data movement (via RDMA) and source/destination address matching (for sends and receives). This can cause some friction when running MPI in highly multi-threaded environments as it complicates progress control. At the same time, it is clear that hardware can support improved runtime work-sharing when integrated properly in an MPI implementation.

2.5 User-Level Work-Sharing

In contrast to runtime work-sharing, application developers consider hybridization from the point of view of exploiting high node-level parallelism across many execution streams created at user-level. The question becomes how well the MPI interface is suited for manipulating chunks of data scattered among several actors, requiring pack/unpack operations and non-desirable sequentialization just to intercommunicate. Requiring each user-level entity to be an independent MPI process to enable communication is a non-starter [18]. Evolving MPI to perform efficiently in a highly parallel section code potentially requires evolution from the present state. We consider such evolutionary steps below.

2.5.1 MPI + X. Most MPI implementations are based on processes and thus start in a sequential execution mode, as launched by the batch manager. Parallel execution is achieved through SPMD execution (i.e., creation of multiple processes) and/or the use of multithreading within an MPI process. In order for MPI to mix with multithreaded shared-memory programming models, the standard provides different levels of thread support and sets a specific level with `MPI_Init(_thread)`. In particular, the `THREAD_MULTIPLE` support level enables MPI calls to be called in parallel by multiple execution streams. Such support is available in most MPI implementations, though robustness and performance can vary. While stability issues are now mostly addressed, there is clearly extra complexity when multiple threads are allowed to call MPI and it will necessarily lead to increased runtime overhead due to the extra locking [33].

Also, there are potential points of conflict between MPI runtime and the user-level work-sharing runtime, as when attempting to overlap communication and computations in a parallel region. Forcing the user to insert MPI calls to make progress is burdensome [6]. Other constraints arise from the fact that each thread in an MPI process has the same logical endpoint (i.e., rank in `COMM_WORLD`). In order to be able to communicate between threads, the user is then required to rely on tags or suitable communicators to ensure message semantic between thread [19]. Other approaches have then been developed to mitigate this constraint.

2.5.2 MPI Sessions. MPI Sessions [22] enable the user to initialize the runtime multiple times in the same UNIX process. The direct consequence is then that concurrent parts of the program can issue MPI calls while manipulating their own instance of the interface thanks to context inheritance through MPI handles. Not only does it separate concerns within the code to allow independent execution, but it also solves a potential race condition when initializing MPI. It, therefore, enables the parallel start of an MPI job mitigating the need for a "master" thread – membership to a communicator is now provided in an "opt-in" fashion instead of implied by the batch manager *process management interface* (PMI) [7].

2.5.3 Endpoints. Another approach is based on the idea of creating MPI processes which are not initially part of the initial `COMM_WORLD`. This could then allow the creation of a communicator encompassing multiple threads per UNIX process. Here, the threads use *communication endpoints* [12] to exchange MPI messages as if they were regular MPI processes. While the MPI rank is unique to a UNIX process, thread-local context passing has been proposed as an array of communicator "handles" combined with an `MPI_Attach` function. This model then enables threads inside a given (OS) process to address the MPI interface as if they were regular MPI processes. However, one may argue that such an interface might be inefficient as it goes against the shared-memory nature of process-local exchanges [18]. Indeed, MPI messages enforce a copy from buffer to buffer whereas a pointer (e.g., ownership passing [16]) could achieve similar transfer in a more efficient manner, thereby avoiding copy and memory overhead [5]. Although discussed in the MPI Forum, such behavior is not supported by the MPI standard.

2.5.4 Finepoints. The last alternative for hybrid support inside MPI is the concept of finepoints [3, 18]. Unlike endpoints, which provide means of enabling regular communications for threads, finepoints provide communication primitives which are to be used by several threads at once. To do so requires persistent communication buffers that are scattered between multiple execution streams and can be triggered to abstract pack and unpack phases in an operation which signals once the last piece of data is available. Consequently, `THREAD_MULTIPLE` is not compulsory to support this interface as the counter trigger acts as a critical section. As far as context handling is concerned, it is simply based on an index determining the destination offset in the buffer and therefore does not involve the MPI rank at the thread level.

2.6 Summary

Our survey of various approaches, current and envisioned, to support increasing parallelism at the node level, suggests potential advantages from combining multiple MPI contexts in shared memory for performance, progress, and network resource factorization. However, there remains lingering question of how MPI either exposes or copes with application parallelism.

Figure 1 illustrates several possible approaches to addressing multiple execution streams with MPI. As discussed above, a common one is to enable `THREAD_MULTIPLE` support, allowing threads to share the same communication rank. By its nature, this solution exposes itself to potential performance issues. Alternatively, MPI Sessions allow process subsets to be addressed potentially concurrently, enabling an `mpi://COMM_WORLD` session to cohabit with an `mpi://rack1` session, for example. In the case of endpoints, threads are able to communicate as independent processes. Finally, finepoints are able to abstract MPI operations issued by threads "serializing" them in a single rank.

It is interesting to observe how these various approaches deal with MPI context. Current MPI implementations usually rely on a process-based global variable or thread-based variables in thread-level storage (TLS) [4]. In finepoints, threads are able to address their target MPI operation with an index for a buffer offset. On the other hand, endpoints rely on MPI handles to transmit the context, in particular, the proposed interface yields an array of communicator handles to abstract calling thread's rank. MPI sessions can be implemented in multiple manners. If rank overloading is precluded (e.g., all members of the sessions are a subset of `COMM_WORLD`), a global variable is sufficient. However, if endpoints are to be created with sessions, the rank will have to be included in the resulting handle to distinguish between execution streams.

The idea of an MPI context (including the rank) being transmitted solely through handles, as we will further explore in the rest of this paper, has several interesting properties. In particular, it allows for the definition of MPI calls as *MPI tasks.* Like one thinks of application tasks, which can be executed (with packed data inputs) independently from the underlying execution stream, it is desirable to have MPI communication operations (invoked with packed parameters) able to be executable independently from the underlying context. The reason for this is twofold. First, it enables the use of MPI communications inside tasks [1], which are by definition independent from a given thread (user-level work-sharing). Second, it allows a thread to execute high-level MPI operation on behalf of another one without context management, information being passed through handles – a process otherwise limited to runtime internals [11]. If MPI communications in tasks have already been

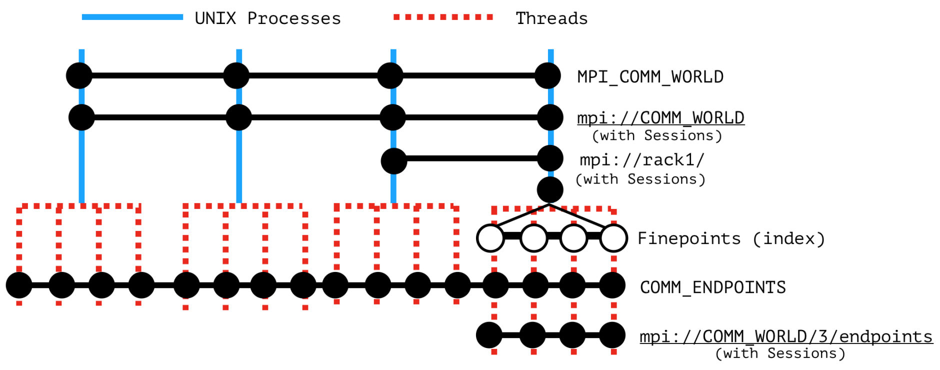


Figure 1: Overview of the various rank configuration for the user-level worksharing configurations of Section 2.5.

studied [29], it lacked the possibility to balance communication workload across multiple MPI ranks. In the rest of the paper, we propose to investigate the latter configuration, considering the design and development of runtime work-sharing with nonblocking collectives in our thread-based MPI runtime MPC.

3 WORKSHARING GENERALIZED REQUESTS

Up to now, we have presented the various hybridization approaches for MPI from both a user-level and a runtime point of view. We argued that saving the MPI rank inside the communicator handle could enable interesting work-sharing capabilities for MPI runtimes collocating ranks (thread-based) or instances (MPI Sessions) inside the same memory space. As our MPC runtime does not have an MPI Session implementation yet, we focus ourselves with what is the closest concept to a task in the MPI standard, *Generalized Requests* (MPI Standard v3.1 - chapter 12). Such requests enable the tracking of arbitrary processing inside an MPI handle. This is done by passing function pointers upon request creation, and using them in the runtime to test the request state (completed or canceled). Unfortunately, in its standard form, MPI does not progress the requests – it has to be done manually by the application, for example by creating a dedicated thread. A related limiting factor is that only the MPI runtime has visibility on idle MPI time.

Non-standard Extended Generalized Request (or EGREQ) [26] can address this limitation by adding a `poll` function pointer at request creation. Main MPI runtimes such as MPICH and OpenMPI[1] do implement EGREQs, this support being mostly motivated by the ROMIO implementation of MPI-IO [30] for request-based operations using EGREQ. Our proposed idea, described in the following sections, is to leverage EGREQ as a means of sharing work between MPI ranks in MPC as it would be possible between Sessions (MPI instances) residing in the same process.

One point that we overlooked until now is how another MPI context can make work progress for potentially another rank. Indeed, our goal is to encapsulate MPI calls inside such a request. Therefore, if we consider the case of work-sharing between MPI instances (ranks for MPC), any MPI operation would inherit its context from the calling thread, issuing operations with the wrong source. The reason is that in the MPC runtime context passing is not implemented through handles, but instead uses a TLS runtime state for a given rank.

Still, we were able to realize such a collaborative request polling by implementing a “disguise” function presented in Listing 1 which simply borrows the whole TLS context from the target MPI rank. As an artifact of MPC relying on a privatizing compiler to stores every global variable inside a dedicated segment (to enable their context switching in the user-level thread scheduler), changing the pointer to this TLS segment immediately transforms a given thread in another one (context-wise) and transitively a given MPI process to another, borrowing its rank.

Listing 1: Context switching interface used in MPC to switch between threads’ contexts

```
/* Become 'target_rank' in 'comm'
if this rank is not local
MPI_ERR_ARG is returned */
int MPCX_Disguise( MPC_Comm comm, int target_rank )
/* Restore original context */
int MPCX_Undisguise()
```

The above feature allows any idle MPI process to steal actions from other MPI processes. The EGREQ polling list of all MPI processes on the node is stored in a shared memory location. If one MPI process does not have any task in its list upon polling the request (e.g., in `MPI_Wait, MPI_Test`), it will start stealing tasks from others. When a task is stolen, the stealing MPI process disguises itself in the target MPI process by changing its root TLS pointer. It can then perform the action encapsulated in the task. Eventually, as soon as the action is finished, the MPI process returns back to its own self by resetting its TLS pointer. We acknowledge the fact that in the near future, context embedding inside MPI handles should make this “disguise” approach irrelevant, as the MPI call parameters would hold the context. Nonetheless, in the absence of MPI sessions in our runtime this simple mechanism (pointer switch) allows us to assess new capabilities of MPI 4.0 in a representative manner.

[1]Since June 2018 PR #5337

4 IMPLEMENTING NONBLOCKING COLLECTIVES WITH MPI TASKS

Now that we introduced how and why we implemented *MPI Tasks* with EGREQs, we propose to experiment with them on the nonblocking Collective (NBC) implementation inside MPC. This MPI feature is of interest as it relies on both (1) overlap to recover the collective and (2) potentially work-shared processing for reduction operations. We compare this approach with the current implementation of NBC inside MPC based on a specifically tailored implementation of libNBC [8, 9, 32].

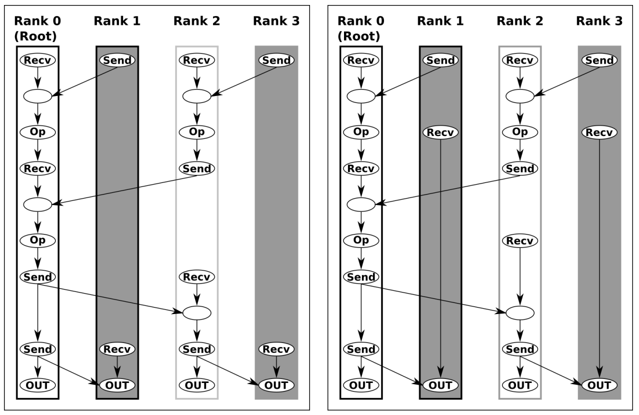
4.1 Reference Implementation using libNBC

The library libNBC [21] has been developed as a first portable implementation to promote the integration of nonblocking collective in the standard. The implementation of these nonblocking collectives in the library can be summarized as follows. A collective communication can be decomposed in a set of point-to-point communications, plus computations if some operation is involved (e.g., MPI_Reduce). The libNBC implements one or several algorithms for each nonblocking collectives. According to the algorithm being performed, given the current rank, the number of MPI processes in the MPI communicator used for the collective, and whether the current rank is the root or not, one can know the exact sequence of point-to-point operation each MPI process has to perform to execute the complete collective communication. For each nonblocking collective call, the libNBC creates a *NBC_Handle* structure containing a *NBC_Schedule* structure, which is the list of point-to-point communications assigned to current MPI process. To allow more overlap, these point-to-point communications are also nonblocking.

The libNBC employs a progress thread per MPI process to be more asynchronous and thereby leverage better performances. Indeed, if MPI calls are nonblocking, it does not mean that they progress asynchronously and therefore it is the calling program which has to *test* progress. The consequence is that embedding additional processing such as NBC in MPI required an explicit dedication of resources. In particular, for libNBC, a thread is triggered by a message to `MPI_COMM_SELF` when starting a new collective.

Despite providing enhanced overlap, a dedicated progress thread per task failed to fully leverage the asynchronous behavior. The arising problems can be illustrated on one of the algorithms for the `MPI_Iallreduce` routine: the tree-based algorithm where the `MPI_Iallreduce` uses the same steps as a tree-based `MPI_Ireduce` followed by a tree-based `MPI_Ibcast`. Figure 2(a) displays a "logical" view of the actions, considering that the MPI process with rank 0 is the root. As the algorithm is depicted as two logical trees, the actions belonging to one stage of the trees are grouped on the same line. However, how the actions are realized at runtime does not strictly follow the tree representation. Indeed, if the first action is performed, there is no reason for an MPI process to wait before starting the next action. The actual actions timeline is depicted in Figure 2(b).

With such an algorithm for `MPI_Iallreduce`, the number of communications is not balanced between MPI processes. Leaf MPI processes have only two routines to call: `Send` to send their input buffer, and `Recv` to receive the final result in their output buffer. Hence, the `Recv` is called right after the `Send` is completed. The



(a) Logical view where each action begins at the according step in the tree (b) Temporal view where each action begins as soon as possible

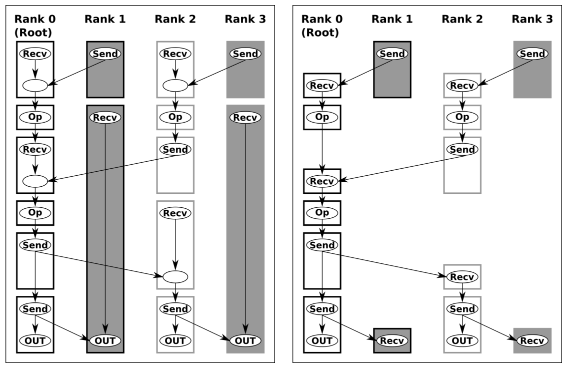
Figure 2: Comparing (MPI_Iallreduce+MPI_Wait) logical and temporal implementation with progress threads.

early start of the latest action prevents efficient scheduling. First, the thread cannot be put to sleep because there is still work to do for the collective and it is already triggered. At last, due to the implementation of libNBC, the progress function cannot work on another nonblocking collective until the current one is done. Hence, the huge gap of inactivity between the first and the last call cannot be filled with communications from another collective. If a thread-based implementation of nonblocking collective asynchronous steps does not allow efficient interleaving of such collective, a task-based implementation of such progression may improve this behavior. For this reason, we developed a new implementation of nonblocking collectives based on *Extended Generic requests*, which can be considered as "MPI tasks".

4.2 Implementing NBCs using EGREQs

To implement this new version of nonblocking collectives, several MPI requests were packed in a list behind a single generic request handler. Dealing with the polling function it processes one operation from the request and marks the collective completed accordingly. If actions to be performed by each MPI process are comparable to the libNBC implementation, using generalized request create a real separation between these actions. These actions can now be seen as independent *MPI tasks* (tasks that perform only one MPI action). Having tasks instead of agglomerate actions in one NBC_Schedule handle by a progress thread should provide a better interleaving capability. However, the basic implementation remains very similar to the thread implementation.

The implementation of the previous `MPI_Iallreduce` algorithm using tasks is represented in Figure 3. On the left-hand side, the algorithm is displayed with tasks without dependencies. For each collective, the same set of actions per MPI process than libNBC is spawned in the form of tasks. These tasks are enqueued in a ready list of tasks. When the progression happens, the tasks are dequeued in a First-In-First-Out (FIFO) order. Hence, for an MPI process, once a task is performed, it automatically dequeues the



(a) (MPI_Iallreduce+MPI_Wait) with tasks (b) (MPI_Iallreduce+MPI_Wait) with tasks and dependencies

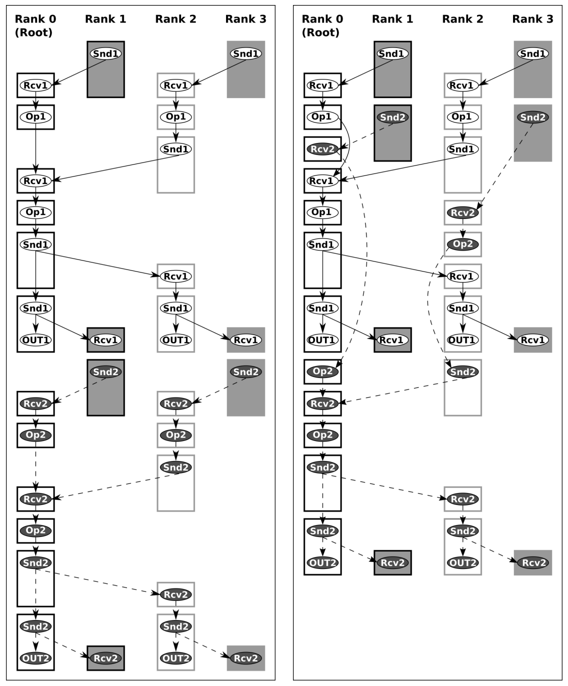
Figure 3: (MPI_Iallreduce+MPI_Wait) algorithm implemented with "tasks" as tree-based Reduce+Broadcast.

next action and starts it. This is the same behavior as with libNBC. Furthermore, also the same as libNBC, it is not possible to interleave the resolution of multiple nonblocking collectives. Since tasks are enqueued and dequeued in order, an MPI process has to perform all the tasks of the first nonblocking collective before reaching the first task of the second nonblocking collective. Hence, the overall behavior of the task-based NBC is relatively similar to the one of the worker-based at the exception that work items can be stolen.

Using tasks with dependencies can leverage interleaving. If such dependencies are difficult to implement in internode context, it is easier on intra-node using shared memory (e.g., either with a thread-based MPI or with shared memory segment). The behavior of the tree-based `MPI_Iallreduce` algorithm with such tasks is depicted in Figure 3(b). Here, an MPI process does not directly launch the next task when one is completed. Instead, it has to wait until all the tasks dependencies (incoming arrows in the Directly Acyclic Graph representing the data flow in the tree) are resolved. Triggering the task actions only when it is relevant creates more idle time on the computing resources (all the empty spaces between tasks in Figure 3(b).

5 IMPLEMENTING NBCS USING EGREQS AND MPI RANK DISGUISE

However, providing idle times is not enough to efficiently schedule MPI tasks. With the tree-based algorithm for `MPI_Iallreduce`, the main problem comes from the amount of actions imbalance between MPI processes. The behavior of two consecutive `MPI_Iallreduce` is represented in Figure 4(a). The two `MPI_Iallreduce` have the same root: MPI process with rank 0. Each MPI process has its own color scheme for the represented tasks: white fill and black border for rank 0, grey fill and block border for rank 1, white fill and grey border for rank 2, and grey fill and grey border for rank 3. To easily identify the actions of each `MPI_Iallreduce`, the actions from the first `MPI_Iallreduce` are with white fill and black labels, while the actions from the second `MPI_Iallreduce` are with dark grey fill and white labels.



(a) (MPI_Iallreduce+MPI_Wait) with tasks (no disguise) (b) (MPI_Iallreduce+MPI_Wait) with tasks and disguise

Figure 4: Two(MPI_Iallreduce+MPI_Wait) algorithms implemented with "tasks" as tree-based Reduce+Broadcast either with or without disguise.

With this tree-based algorithm, all odd-ranked MPI processes have only two tasks to perform: `Send` to send their input buffer, and `Recv` to receive the final result in their output buffer. The gap between these two tasks of the first `MPI_Iallreduce` is used to perform the first operations of the second `MPI_Iallreduce`. This can be seen in the Figure with the second "Snd" dark grey task directly following the first "Snd" tasks at the odd ranks. The interleaving allows reducing the logical end time of this second operation. But the last tasks of the two `MPI_Iallreduce` cannot be performed earlier due to their dependencies. In the presented case, the main bottleneck is the root MPI process. In comparison to the odd-ranked MPI process have to call much more routines, with log(n) (`Recv` + `Op`) and log(n) `Send` (n being the number of MPI processes involved in the collective communication).

The best solution would be for the inactive MPI processes to help perform the tasks of the loaded MPI processes, such as the root. In this case, represented in the right-hand side of Figure 4, the most inactive odd ranks perform tasks from the even ranks. After the second "Snd" dark grey task, scheduled in the same position as in the left-hand side, we can observe for each odd rank the scheduling of tasks from other MPI processes (white fill black border tasks from rank 0 on rank 3, and white fill grey border tasks from rank 2 on rank 1). This task stealing provides better load-balancing and provides an earlier logical ending time. The use of our proposed feature `MPIX_Disguise` allows this behavior. Once an MPI process

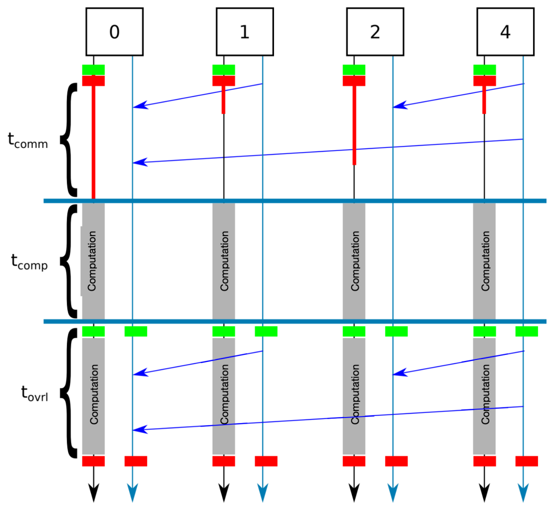


Figure 5: Illustration of the main timings considered when characterizing nonblocking communications.

is inactive, it can look to the global list of pending tasks and disguise himself as the MPI process which should perform the next task.

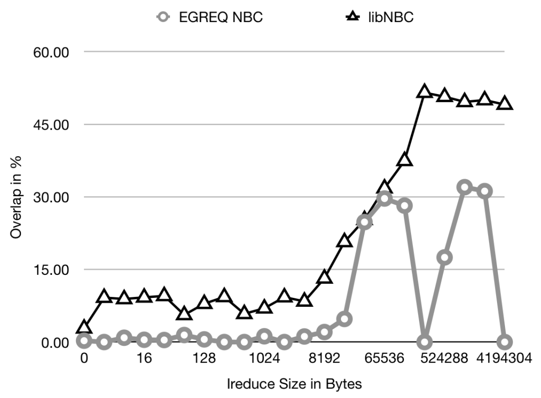
6 TASK ORIENTED NBC PERFORMANCE RESULTS

We implemented collaborative polling of EGREQ with support for disguise. This allows all threads running various MPI processes on the same node to opportunistically progress requests including MPI calls. The results we present here do not include the support for dependencies, triggering the receive tasks only when the corresponding (local) send is done hence some tasks may block on data dependency as depicted in Figure 3(a). Hence, the scheduling we can achieve is less efficient than the one described in Figure 4(b). Still, with the implementation we measure, we have support for stealing tasks between MPI processes, including processes of different ranks running on the same node.

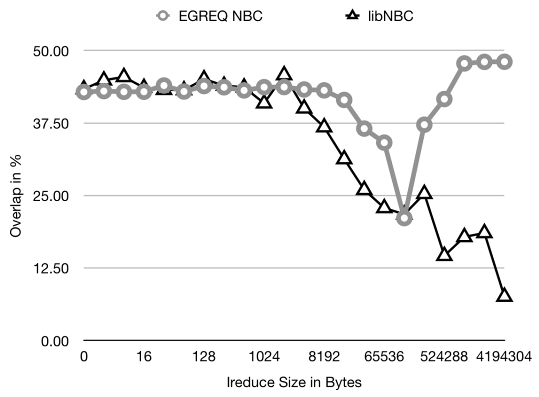
Tests were done on a cluster composed of Bull Sequana nodes each with two Intel Xeon Platinum 8168 processors (Skylakes) with 24 cores (48 threads) for a total of 48 cores per node. All tests were done without using hyper-threading. As far as the interconnect is concerned, it relies on Mellanox MT27700 ConnectX-4 Host-Channel Adapters (Infiniband EDR).

6.1 Measuring Nonblocking Collectives

Figure 5 presents the main timings to be considered when measuring nonblocking communication performance. First, we consider the "pure" time of the communication or T_{comm} which represents the time of a nonblocking collective when directly waited (no computation). Second, we consider T_{comp} which is the time of the computation to be recovered, usually, this time is chosen to be approximately the time of the communication. Eventually, T_{ovrl} represents the overall time associated with the asynchronous communication *overlapped* with computation. In this last case, an overlap ratio can be calculated as such that T_{ovrl} is lower than the sum



(a) IReduce with 2 ranks



(b) IReduce with 48 ranks

Figure 6: Comparison of EGREQ-based IReduce with the LibNBC one when running in thread-based on 2 and 48 MPI processes (one node in Thread-based).

$T_{comm} + T_{comp}$, communications being recovered. In the rest of this section, we will consider these metrics in the light of two benchmarks, the classical IMB and a second benchmark which allows us to fix the computation time in order to characterize overlap.

6.2 IMB Results

We made a first set of measurements using the Intel MPI Benchmarks (IMB) in order to characterize potential gains associated with our method, with respect to the currently available implementation in MPC, based on libNBC. For a performance comparison with respect to other MPI runtimes implementation one may refer to our previous work on the subject [8, 9, 32].

If we now consider the IReduce operation presented in Figure 6, results are more at the advantage of EGREQ in terms of overlap. The reason for this is that the tree-based reduce operation starts with important parallelism (all buffers are to be processed) and that it involves computation and communication yielding more work to be stolen using MPI tasks. Hence, while with two MPI processes, EGREQ does not outperform libNBC, see Figure 6(a), the overlap is greatly improved with 48 ranks as shown in Figure 6(b). This shows that runtime-level work-sharing can provide some benefits in terms of overlap and, therefore, that patterns gathering multiple ranks (thread-based) or multiple instances of MPI (sessions) can open some common optimization schemes – improving runtime parallelism. However, as the IMB do adapt the computation time to maintain a 50/50 ratio between communication and computation, overlap alone is not sufficient to measure nonblocking collective improvements.

In fact, if we look at Figure 7, which presents this time the total excecution time (T_{ovrl}), we can see that our EGREQ model performs generally badly with respect to the initial implementation. It appears that our proof-of-concept integration of EGREQ tasks in the runtime is not sufficient. In particular, such requests are progressed solely when each thread is individually processing an operation belonging to such request. In particular, a Recv operation in a EGREQ NBC can poll for other operations when waiting. However, once a rank has done its contribution it does not remain in the progress. A lower-level scheduler would solve this issue by providing a constant progress to MPI-related tasks, this is part of future work. As a result, Reduce behaves poorly in our first implementation. However, it is not the case for all collectives.

For example, if we consider Figure 8, we can see that again the EGREQ model efficiency is dependent on the availability of a sufficient number of threads to enable collaborative progress, hence being slower for smaller number of threads. However, when the number of collocated MPI processes rises, performance are comparable, and sometimes even slightly improved.

6.3 Measuring Overlap

In a second time, we relied on a custom benchmark where computing time can be fixes dynamically in order to measure overlap. As we have seen previously, it is not trivial to qualify gains with respect to nonblocking collectives solely with overlap ratio. We therefore relied on a fixed computation size that we have set to the EGREQ collective time, allowing us to simulate the same computing payload. This benchmark has been used in previous work [8] to assess MPC's NBC performance. In order to retain a metric which is simple to analyze, we present the T_{ovrl} time which is the time to complete the collective when overlapped with computation.

As presented in Figure 9, our method provides only small gains on the overall execution time, with slightly improved performance on larger collectives. Ibcast's tree-based algorithm yields better results than the linear Igather. Such algorithms were retained to match the ones which are used as reference in the libNBC used in MPC. Overall, it is clear that spatially diffuse collective seem to be better candidates for optimization. This approach, despite not yielding significant improvements yet, opens the way for a more collaborative vision of MPI runtime internals in a context where tasks are about to be prominent. We expect that a more integrated approach with an unified schedule should harvest the potential gains we identified in previous measurements, aspect that we intend to evaluate in future work.

7 CONCLUSION

We started this paper with a relatively general discussion of what tracks are being (or were) explored with respect to MPI hybridization. In particular, we identified two main tracks, one minimizing the end-user impact by "hiding" efforts in the runtime, that we denoted runtime-level work-sharing. And the other which exposes new means of describing parallelism to the application, requiring changes in the code and that we called user-level work-sharing. This done, we posed the question of what a rank is in MPI, and transitively leading us to ask where the context is usually managed in MPI. In fact, ranks are generally linked to a global variable and therefore implicitly tied to the current UNIX process. In thread-based MPIs such as in MPC, this context is stored in a TLS which is context-switched between user-level threads. When considering endpoints, the communicator handle had to be duplicated to "shadow" the parent process context. MPI Sessions do take this approach one step further by embedding the context in handles so that global dependencies for a given MPI operation disappear as long as the correct handles are passed.

This is an important change for MPI which can now run for example inside task-based models which may encompass multiple instances (or ranks) running pieces of work on multiple threads. It also opens the way for the collocation of multiple MPI instances in the same process. In complement of this user-level parallelism improvement, we argue that such patterns enable runtime optimization. To illustrate this, we reproduced an environment approaching one of the sessions inside our thread-based MPI runtime MPC, enabling context sharing through "disguisement". Being able to have one rank working for another, we show that there is room for improvements with asynchronous operations. We used NBC as a first test-case, comparing our progress-thread implementation based on libNBC with one relying on MPI tasks and Extended Generic Requests. This transitively shows that Sessions which won't need such context switching should be a candidate for similar optimizations.

Overall, the propagation of the MPI context through handles implied by sessions is an important evolution for MPI as it will provide a clear context to each MPI call, e.g. its parameters. Indeed, recent evolution in shared-memory programming models including tasks as, for example, in OpenMP broke the correlation between the code to be executed and the thread running it. Hence, running MPI in such context became more and more complex due to the implied global context stored in the UNIX process. One can see that if now MPI is broken down in sub-components (ranks or Sessions) living in the same process the issues is even more obvious as the context is more than a global process-id. Yet, our paper shows that there is a potential for runtime gain by collocating multiple runtimes as such, a common view from thread-based MPI could be semantically transposed to Sessions enabling all runtime-sharing and collaborative polling patterns. All this depends on the fact that the MPI context is fully stored in the handle and that the underlying runtime does not rely, for example, on a translation table (e.g. a sub-communicator of `COMM_WORLD`) to implement sessions as otherwise the "shadow context" would still be at play. In addition, this support is required to implement "endpoints" on top of sessions as such identifier would not be transposable in `COMM_WORLD`. Our potential runtime gains advocate for the later implementation.

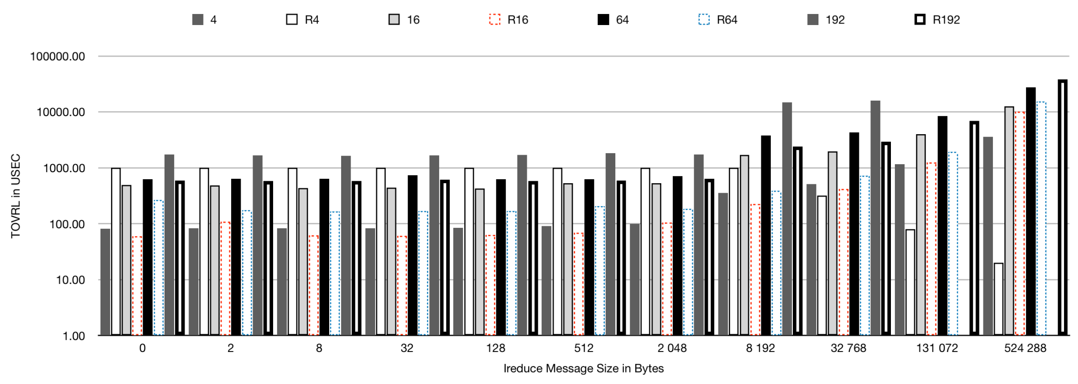


Figure 7: Comparisons of TOVRL for Ireduce, measured with IMB for a run on 4 nodes with a varying number of MPI processes (one UNIX process per node). R means (R)eference for the inital libNBC, numbers are the number of MPI Processes.

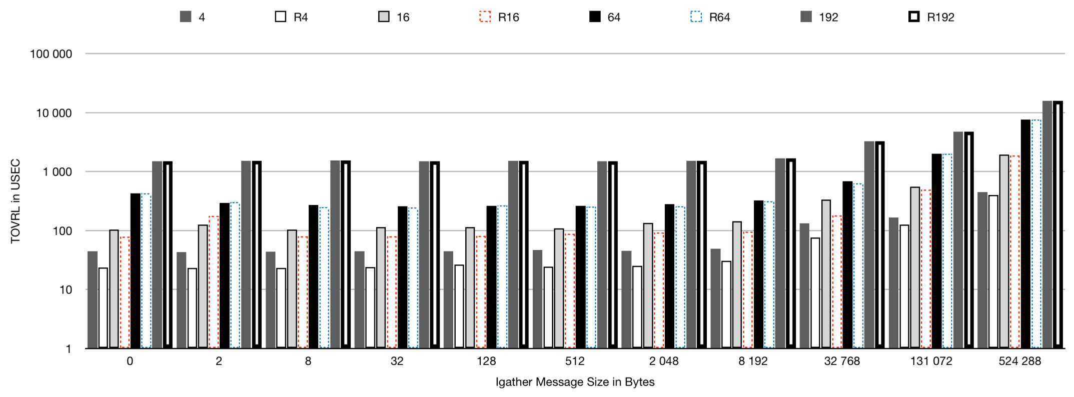


Figure 8: Comparisons of TOVRL for Igather, measured with IMB for a run on 4 nodes with a varying number of MPI processes (one UNIX process per node). R means (R)eference for the inital libNBC, numbers are the number of MPI Processes.

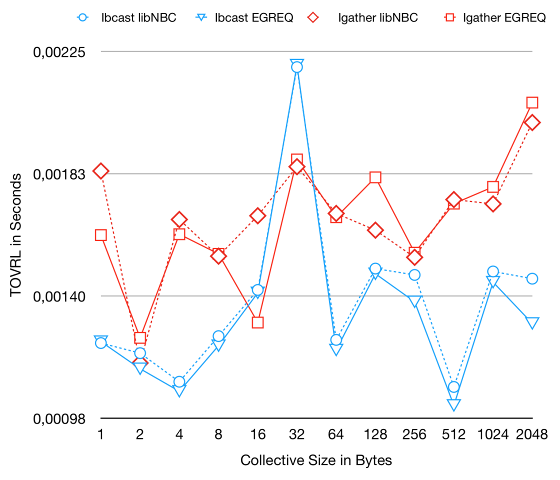


Figure 9: T_{ovrl} for Igather and Ibcast using our custom benchmark. Test was run on 192 MPI Processes, 4 UNIX processes on 4 Nodes.

8 FUTURE WORK

There is still a lot of room for improvement in our implementation. First, expressing dependencies would improve scheduling. Then, as mentioned in the paper, our disguise approach specific to a thread-based MPI runtime can be transposed as collocated Sessions. It is then of high interest to implement sessions to evaluate this gain between instances (instead of between ranks) to assess that these two configurations are comparable. Eventually, in the continuity of MPI tasks seen as context independent MPI calls, such operations should be supported by MPI. In fact, Sessions will soon provide tightly coupled communication groups with a new dynamic but they cannot communicate yet between each other. Existing MPI calls could be complemented through the pendant of MPI tasks as communications, i.e. Remote Procedure Calls (RPCs) or Active Messages (AM) [20, 34, 35] in MPI semantics. This would open one-sided, execution context-free communications between sessions currently not in MPI despite fitting interesting scenarios (in-situ, steering, client-server, tools, ...).

REFERENCES

[1] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. 2009. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems* 20, 3 (March 2009), 404–418. https://doi.org/10.1109/TPDS.2008.105

[2] Brian Barrett, Ronald B. Brightwell, Ryan Grant, Kevin Pedretti, Kyle Wheeler, Keith D. Underwood, Rolf Riesen, Arthur B. Maccabe, Trammel Hudson, and Scott Hemmert. 2017. The Portals 4.1 Network Programming Interface. (4 2017). https://doi.org/10.2172/1365498

[3] David E. Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E. Grant, Thomas Naughton, Howard P. Pritchard, Martin Schulz, and Geoffroy R. Vallee. [n. d.]. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience* 0, 0 ([n. d.]), e4851. https://doi.org/10.1002/cpe.4851 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4851 e4851 cpe.4851.

[4] Jean-Baptiste Besnard, Julien Adam, Sameer Shende, Marc Pérache, Patrick Carribault, Julien Jaeger, and Allen D. Maloney. 2016. Introducing Task-Containers As an Alternative to Runtime-Stacking. In *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI 2016)*. ACM, New York, NY, USA, 51–63. https://doi.org/10.1145/2966884.2966910

[5] Jean-Baptiste Besnard, Allen Malony, Sameer Shende, Marc Pérache, Patrick Carribault, and Julien Jaeger. 2015. An MPI Halo-Cell Implementation for Zero-Copy Abstraction. In *Proceedings of the 22Nd European MPI Users' Group Meeting (EuroMPI '15)*. ACM, New York, NY, USA, Article 3, 9 pages. https://doi.org/10.1145/2802658.2802669

[6] Antoine Capra, Patrick Carribault, Jean-Baptiste Besnard, Allen D. Malony, Marc Pérache, and Julien Jaeger. 2017. User Co-scheduling for MPI+OpenMP Applications Using OpenMP Semantics. In *Scaling OpenMP for Exascale Performance and Portability*, Bronis R. de Supinski, Stephen L. Olivier, Christian Terboven, Barbara M. Chapman, and Matthias S. Müller (Eds.). Springer International Publishing, Cham, 203–216.

[7] Ralph H. Castain, David Solt, Joshua Hursey, and Aurelien Bouteiller. 2017. PMIx: Process Management for Exascale Environments. In *Proceedings of the 24th European MPI Users' Group Meeting (EuroMPI '17)*. ACM, New York, NY, USA, Article 14, 10 pages. https://doi.org/10.1145/3127024.3127027

[8] Alexandre Denis, Julien Jaeger, Emmanuel Jeannot, Marc Pérache, and Hugo Taboada. 2018. Dynamic Placement of Progress Thread for Overlapping MPI Non-blocking Collectives on Manycore Processor. In *Euro-Par 2018: Parallel Processing*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). Springer International Publishing, Cham, 616–627.

[9] Alexandre Denis, Julien Jaeger, and Hugo Taboada. 2019. Progress Thread Placement for Overlapping MPI Non-blocking Collectives Using Simultaneous Multithreading. In *Euro-Par 2018: Parallel Processing Workshops*, Gabriele Mencagli, Dora B. Heras, Valeria Cardellini, Emiliano Casalicchio, Emmanuel Jeannot, Felix Wolf, Antonio Salis, Claudio Schifanella, Ravi Reddy Manumachu, Laura Ricci, Marco Beccuti, Laura Antonelli, José Daniel Garcia Sanchez, and Stephen L. Scott (Eds.). Springer International Publishing, Cham, 123–133.

[10] S. Derradji, T. Palfer-Sollier, J. Panziera, A. Poudes, and F. W. Atos. 2015. The BXI Interconnect Architecture. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 18–25. https://doi.org/10.1109/HOTI.2015.15

[11] Sylvain Didelot, Patrick Carribault, Marc Pérache, and William Jalby. 2014. Improving MPI communication overlap with collaborative polling. *Computing* 96, 4 (01 Apr 2014), 263–278. https://doi.org/10.1007/s00607-013-0327-z

[12] James Dinan, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. 2013. Enabling MPI Interoperability Through Flexible Communication Endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting (EuroMPI '13)*. ACM, New York, NY, USA, 13–18. https://doi.org/10.1145/2488551.2488553

[13] Dan Holmes et al. 2019. MPI Session working group wiki. https://github.com/mpiwg-sessions/sessions-issues/wiki

[14] Mario Flajslik, James Dinan, and Keith D. Underwood. 2016. Mitigating MPI Message Matching Misery. In *High Performance Computing*, Julian M. Kunkel, Pavan Balaji, and Jack Dongarra (Eds.). Springer International Publishing, Cham, 281–299.

[15] Andrew Friedley, Greg Bronevetsky, Torsten Hoefler, and Andrew Lumsdaine. 2013. Hybrid MPI: Efficient Message Passing for Multi-core Systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 18, 11 pages. https://doi.org/10.1145/2503210.2503294

[16] Andrew Friedley, Torsten Hoefler, Greg Bronevetsky, Andrew Lumsdaine, and Ching-Chen Ma. 2013. Ownership Passing: Efficient Distributed Memory Programming on Multi-core Systems. *SIGPLAN Not.* 48, 8 (Feb. 2013), 177–186. https://doi.org/10.1145/2517327.2442534

[17] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. 1996. MPI-2: Extending the message-passing interface. In *European Conference on Parallel Processing*. Springer, 128–135.

[18] Ryan Grant, Anthony Skjellum, and Purushotham V Bangalore. 2015. *Lightweight threading with MPI using Persistent Communications Semantics.* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[19] Torsten Hoefler, Greg Bronevetsky, Brian Barrett, Bronis R. de Supinski, and Andrew Lumsdaine. 2010. Efficient MPI Support for Advanced Hybrid Programming Models. In *Recent Advances in the Message Passing Interface*, Rainer Keller, Edgar Gabriel, Michael Resch, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–61.

[20] Torsten Hoefler, Salvatore Di Girolamo, Konstantin Taranov, Ryan E. Grant, and Ron Brightwell. 2017. sPIN: High-performance Streaming Processing In the Network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 59, 16 pages. https://doi.org/10.1145/3126908.3126970

[21] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. 2007. Implementation and Performance Analysis of Non-blocking Collective Operations for MPI. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*. ACM, New York, NY, USA, Article 52, 10 pages. https://doi.org/10.1145/1362622.1362692

[22] Daniel Holmes, Kathryn Mohror, Ryan E. Grant, Anthony Skjellum, Martin Schulz, Wesley Bland, and Jeffrey M. Squyres. 2016. MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale. In *Proceedings of the 23rd European MPI Users' Group Meeting (EuroMPI 2016)*. ACM, New York, NY, USA, 121–129. https://doi.org/10.1145/2966884.2966915

[23] Daniel John Holmes. 2012. McMPI–a managed-code message passing interface library for high performance communication in C. (2012). http://hdl.handle.net/1842/7732

[24] Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jai Dayal, Pavan Balaji, and Yutaka Ishikawa. 2018. Process-in-process: Techniques for Practical Address-space Sharing. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18)*. ACM, New York, NY, USA, 131–143. https://doi.org/10.1145/3208040.3208045

[25] Chao Huang, Orion Lawlor, and L. V. Kalé. 2004. Adaptive MPI. In *Languages and Compilers for Parallel Computing*, Lawrence Rauchwerger (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 306–322.

[26] Robert Latham, William Gropp, Robert Ross, and Rajeev Thakur. 2007. Extending the MPI-2 Generalized Request Interface. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Franck Cappello, Thomas Herault, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 223–232.

[27] Stas Negara, Gengbin Zheng, Kuo-Chuan Pan, Natasha Negara, Ralph E. Johnson, Laxmikant V. Kalé, and Paul M. Ricker. 2011. Automatic MPI to AMPI Program Transformation Using Photran. In *Euro-Par 2010 Parallel Processing Workshops*, Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 531–539.

[28] Marc Pérache, Hervé Jourdren, and Raymond Namyst. 2008. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Euro-Par 2008 – Parallel Processing*, Emilio Luque, Tomàs Margalef, and Domingo Benítez (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 78–88.

[29] Kevin Sala, Xavier Teruel, Josep M. Perez, Antonio J. Peña, Vicenç Beltran, and Jesus Labarta. 2019. Integrating blocking and non-blocking MPI primitives with task-based programming models. *Parallel Comput.* 85 (2019), 153 – 166. https://doi.org/10.1016/j.parco.2018.12.008

[30] S. Seo, R. Latham, J. Zhang, and P. Balaji. 2015. Implementation and Evaluation of MPI Nonblocking Collective I/O. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 1084–1091. https://doi.org/10.1109/CCGrid.2015.81

[31] M. Si, A. J. Peña, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa. 2015. Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures. In *2015 IEEE International Parallel and Distributed Processing Symposium*. 665–676. https://doi.org/10.1109/IPDPS.2015.35

[32] Hugo Taboada. 2018. *Recouvrement des Collectives MPI Non-bloquantes sur Processeur Manycore.* Ph.D. Dissertation. http://www.theses.fr/2018BORD0365 Thèse de doctorat dirigée par Jeannot, Emmanuel et Denis, Alexandre Informatique Bordeaux 2018.

[33] R. Thakur, W. Gropp, Mathematics, Computer Science, and Univ. of Illinois. 2009. Test suite for evaluating performance of multithreaded MPI communication. *Parallel Comput.* 35, 12 Dec. 2009 (12 2009). https://doi.org/10.1016/j.parco.2008.12.013

[34] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. 2010. AM++: A Generalized Active Message Framework. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. ACM, New York, NY, USA, 401–410. https://doi.org/10.1145/1854273.1854323

[35] X. Zhao, D. Buntinas, J. Zounmevo, J. Dinan, D. Goodell, P. Balaji, R. Thakur, A. Afsahi, and W. Gropp. 2013. Toward Asynchronous and MPI-Interoperable Active Messages. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. 87–94. https://doi.org/10.1109/CCGrid.2013.84