Prototyping the Reference Implementation of Persistent Non-Blocking Collective Communication in MPI

by

Ian Bradley Morgan

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
July 10, 2017

Keywords: MPI, collective communication, persistence, nonblocking, optimized algorithm

Copyright 2017 by Ian Bradley Morgan

Approved by

Anthony Skjellum, Chair, Charles D. McCrary Eminent Scholar Endowed Chair Professor of Computer Science and Software Engineering, Auburn University
Xiao Qin, Professor, Department of Computer Science and Software Engineering, Auburn University
Purushotham V. Bangalore, Professor, Department of Computer and Information Sciences, University of Alabama at Birmingham

Abstract

This research focuses on the design, prototype implementation and standardization status of persistent non-blocking collective operations for the Message Passing Interface (MPI) standard, through functional extension of OpenMPI, an open-source MPI implementation. MPI is an integral tool in the field of parallel computing, providing fundamental communication techniques for large scale computations. Optimization techniques such as persistence (planned transfers) and non-blocking operations (overlapping communication and computation) have been present in point-to-point communication since MPI-1. Similarly, collective operations (communication patterns among groups of processes) have been historically available as a tool to promote convenience and efficiency. More recently, the adoption of non-blocking collective operations into the MPI standard has offered further performance opportunity. For data-parallel and regular computations with fixed communication patterns, more optimization potential can be revealed through the application of persistence to these non-blocking collective functions, yet to be included in the MPI standard.

Persistent operations allow MPI implementations to make intelligent choices about algorithm and resource utilization once, and amortize the decision cost across many uses in a long-running program. This research presents the first prototype implementation of persistence applied to non-blocking collective operations, and demonstrates the functionailty of the corresponding APIS. Early performance results of this implementation and examples illustrating the potential performance enhancements for such operations are presented. Further enhancement of the current implementation prototype, and additional opportunities to enhance performance through the application of these new APIs comprise future work.

Acknowledgments

I would like to offer sincere appreciation to my research advisor, Dr. Anthony Skjellum for the affordance of opportunity, and his assistance and support in this effort. Thanks to my committee members Dr. Puri Bangalore and Dr. Xiao Qin for their perspective, guidance, and instruction on parallel computing. I would also like to express gratitude to Dr. Daniel Holmes of EPCC for his advice, leadership, and invaluable code review. To the OpenMPI developer community, a warm thank you for your welcoming and encouraging spirit. A very special thanks to Shane Farmer for his technical and programming advice. To my wife Theresa, son Jackson, and daughter Goldie, love and thanks for your motivation and support. Finally, an ardent show of appreciation to the Auburn University Office of Information Technology, specifically the Hopper Cluster system administrators, for their support and camaraderie.

Table of Contents

Abstract . . . ii

Acknowledgments . . . iii

List of Figures . . . vii

1 Introduction . . . 1

1.1 Overview . . . 1

1.2 Background . . . 2

1.2.1 Collective Operations . . . 2

1.2.2 Non-blocking Communication . . . 3

1.2.3 Persistence . . . 4

1.3 Motivations and Use Cases . . . 5

2 Design . . . 7

2.0.1 Initialization . . . 7

2.0.2 Starting . . . 10

2.0.3 Completion . . . 12

2.0.4 Freeing . . . 12

3 Implementation . . . 14

3.1 Requests and Scheduling . . . 14

3.2 Initialization . . . 17

3.3 Starting . . . 18

3.4 Development Methods and Tools . . . 21

3.4.1 OpenMPI . . . 21

3.4.2 Open MPI Implementation Model . . . 22

3.4.3 The Modular Component Architecture . . . 23

3.4.4 Component Development 25
3.4.5 Source Control 26
3.4.6 Software Tools 26
3.4.7 LibNBC 27
3.4.8 Build Process 27
4 Results and Discussions 29
4.1 Benchmarking 29
4.2 Overhead 31
4.3 Crossover 32
5 Future Work 45
5.1 Point-to-Point Persistence 45
5.2 Non-Algorithmic Optimizations 45
5.3 Extension for MPI Endpoints 47
5.4 Other Future Work 48
6 Conclusion 49
7 Appendix A: APIs 50
7.1 Persistent Barrier Synchronization 50
7.2 Persistent Broadcast 51
7.3 Persistent Gather 53
7.4 Persistent Scatter 57
7.5 Persistent Gather-to-all 61
7.6 Persistent All-to-All Scatter/Gather 65
7.7 Persistent Reduce 71
7.8 Persistent All-Reduce 73
7.9 Persistent Reduce-Scatter with Equal Blocks 75
7.10 Persistent Reduce-Scatter 77
7.11 Persistent Inclusive Scan 79

7.12 Persistent Exclusive Scan 81
7.13 Persistent Neighborhood Gather 83
7.14 Persistent Neighborhood Alltoall 87
Bibliography 94

List of Figures

1.1 Collective Operation Diagrams 3

1.2 Sample MPI Program Code: Blocking Send vs. Non-Blocking Send 4

1.3 Sample MPI Program Code: Non-Blocking Send vs. Persistent Send 5

3.1 PNBC Schedule Progress Functions (pnbc.c) 15

3.2 PNBC Relevant Data Structures 16

3.3 MPIX_Start Functions 19

3.4 Sample MPI Program Code: Non-Blocking Send vs. Persistent Send 20

3.5 OpenMPI Abstraction Model[17] 22

3.6 Modular Component Architecture Conceptual Model 23

3.7 OpenMPI Source: Locations of Interest 24

3.8 Forcing Specific Component Selection at Runtime 25

3.9 LibPNBC Component Source Files 25

3.10 Initial Component Re-identification of LibNBC to LibPNBC 27

3.11 General OpenMPI Build Pattern 28

3.12 Environment Settings for Custom OpenMPI Installation 28

4.1 Pseudocode for NBCBench time measurements 30

4.2 Pseudocode for PNBCBench time measurements 30

4.3 Initialization Overhead and Test Progress 32

4.4 Operation Latency Outliers 33

4.5 Allgather Operation Benchmark Results 34

4.6 Allreduce Operation Benchmark Results 35

4.7 Alltoall Operation Benchmark Results 36

4.8 Alltoallv Operation Benchmark Results 37

4.9 Bcast Operation Benchmark Results 38

4.10 Gather Operation Benchmark Results 39

4.11 Gatherv Operation Benchmark Results 40

4.12 Reduce Operation Benchmark Results 41

4.13 Reduce_scatter Operation Benchmark Results 42

4.14 Scatter Operation Benchmark Results 43

4.15 Scatterv Operation Benchmark Results 44

Chapter 1

Introduction

1.1 Overview

The pursuit of high performance, efficiency, and low overhead are essential in the field of research computing. As an integral, widely employed component of the research computing stack, the Message Passing Interface (MPI) and its corresponding implementations offer a number of techniques that enable parallel applications to achieve these goals. Traditional methods that support increased performance include the use of non-blocking communication (the overlap of computation and communication) and collective operations (communication patterns among groups of processes) which offer the potential for increased efficiency and lower overhead. More recently, the introduction of non-blocking collective operations in MPI-3 provided even further opportunity for efficiency, by combining these two approaches. It is also conceivable that another traditional method, persistence (the reuse of arguments) can be added as a way to shorten the critical path of MPI non-blocking collective operations,vimproving the potential efficiency of parallel applications.

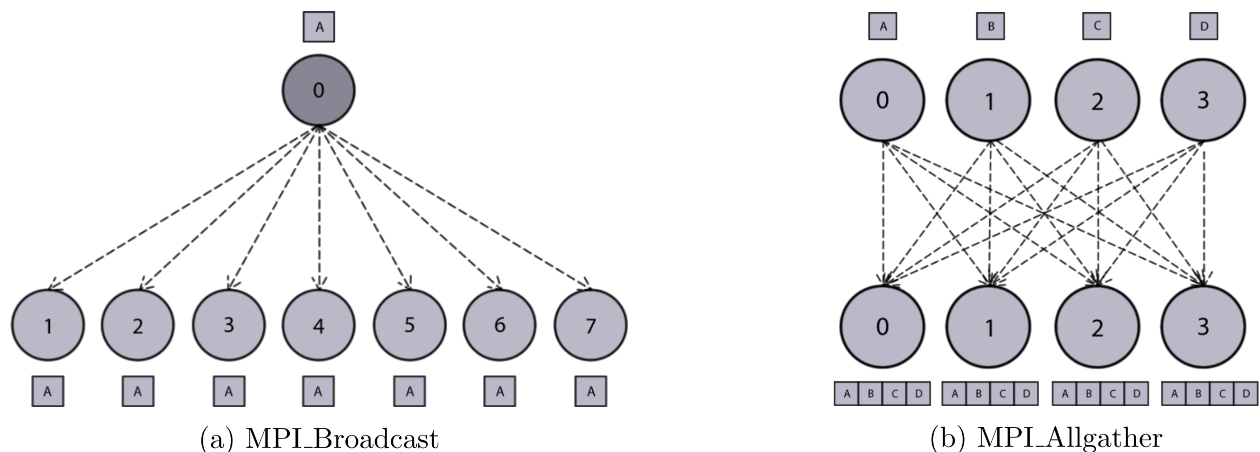
In this effort, new functionality is being demonstrated; non-blocking collective communication that is also persistent, in complement to on-going proposals to the MPI Forum for potential inclusion in the MPI-4 standard. Therefore, this work serves two purposes: to comprise the required demonstration of functionality needed for full consideration of the operations by the Forum, and, second, the basis for a series of implementation iterations leading to practical inclusion of these functions in one or more implementations of MPI, starting with OpenMPI.

1.2 Background

The Message Passing Interface standard (MPI), initially drafted in 1993[5], defines a set of fundamental communication principles for parallel computing. In its most basic form, much like computer networking, this communication is the sharing of data between two endpoints over a common shared medium. Unlike networking, which takes place at lower modular levels, the endpoints involved in MPI communication are software processes, which offer much more computational flexibility, but also require a more dynamic set of communication capabilities. A prevalent set of these features includes collective operations, non-blocking communication, and persistent requests.

1.2.1 Collective Operations

All basic communication in MPI takes place in a point-to-point approach, that is a single sending endpoint sends a message to a single receiving endpoint over a shared medium. A separate descriptor, a communicator, defines a boundary for a collection of these endpoints (processes) for which communication can take place. As common patterns of communication emerge among these processes, it becomes convenient, if not necessary, to define these as operations of their own. These common operational patterns are known as collectives– a defined group of point-to-point operations that replicate certain communication patterns among processes. These patterns are typically represented in code as communication scheduling algorithms, where a set of point-to-point operations are grouped together into a data structure best described as a communication schedule [10]. Common forms include broadcast, reduce, or scatter, where certain participating processes will initiate send or receive calls based on the desired distribution and processing of data among them.



(a) MPI_Broadcast

(b) MPI_Allgather

Figure 1.1: Collective Operation Diagrams

1.2.2 Non-blocking Communication

In point-to-point communication, one endpoint may transmit data to another, and then block any subsequent program instructions until the other process has received, processed, and acknowledged the transmission. This method of blocking communication, while convenient and reliable, introduces inefficiencies where independent computation is possible during message transmission. Because even slight inefficiencies in parallel and iterative programming can lead to significant latency as a program scales, a method for initiating communication over a separate communication thread and returning immediately to the main program is desirable. This non-blocking method of communication was defined for point-to-point operations in MPI-1 [5], and was eventually introduced for collective operations in MPI-3 [6]. In comparison to blocking operations, non-blocking communication requires an additional element, the communication request object, whereby the status of the communication (among other things) can be tracked.

Listing 1.1: Blocking Send

```
if(rank == 0) {
  MPI_Send(&buf,count,MPI_INT,1,1,
     comm);
  // block until return
  do_work();
}
```

Listing 1.2: Non-blocking Send

```
MPI_Request req;
if(rank == 0) {
   MPI_Isend(&buf,...,&req);
   // returns immediately
   do_work();
   MPI_Wait(&req,
      MPI_STATUS_IGNORE);
}
```

Figure 1.2: Sample MPI Program Code: Blocking Send vs. Non-Blocking Send

1.2.3 Persistence

It is often observed that, as non-blocking operations are used in iterative fashion, the arguments passed to their corresponding functions are used repeatedly with little or no modification. In the case of pure non-blocking MPI operations, the request argument's associated data, or even the request object itself, is invalidated or released in memory upon return from its corresponding operation, requiring (at minimum) a reassignment or (maximally) full reallocation for each loop execution. Because of the potential size and complexity of the communication request object, and the consequent overhead involved in its allocation and reuse, persistent requests were introduced in MPI-1 for point-to-point non-blocking operations [16]. A persistent request is created once, either before loop execution, or (only) on the first iteration, and then used repeatedly without the need to reallocate for subsequent operations, allowing for reduced overhead between the process and its communication channel [16].

The set of features discussed in this section: collective operations, non-blocking communication, and persistent communication request objects, are the building blocks of a parallel programming capability for MPI: persistent non-blocking collectives, discussed and implemented in this research.

Listing 1.3: Non-blocking Send

```
MPI_Request req;
for (i=0; $i<$max; i++) {
   MPI_Isend(&buf,count,MPI_INT
      ,1,1,comm,&req);
   do_work();
   MPI_Wait(&req,
      MPI_STATUS_IGNORE);
   // request buffers released
}
```

Listing 1.4: Persistent Send

```
MPI_Request preq;
MPI_Isend_init(&buf,..., &preq);
for (i=0; $i<$max; i++) {
   MPI_Start(&preq);
   do_work();
   MPI_Wait(&preq,
      MPI_STATUS_IGNORE);
   // request buffers held
}
```

Figure 1.3: Sample MPI Program Code: Non-Blocking Send vs. Persistent Send

1.3 Motivations and Use Cases

The primary objective of persistence is to exploit temporal locality present in highly iterated operations; that is, those operations in which the same arguments are passed consistently among all processes participating in the operation (most often, the communicator group's scope). Rather than recreating an MPI request handle each time, a single request object is created and reused for each loop execution.

Gallardo et al. [3, 7] offer convincing evidence that MPI implementations are not taking the best decisions for collective operations as a function of group size, message length, and operation. They show that the just-in-time choices can be significantly far from optimal, in some cases for short transfer lengths, the case during which decision overhead would need to be shortest with a current, non-persistent MPI collective. These performance deficiencies motivate providing specific means for programmers to identify properties and reuse of MPI collectives that would support cost-effective algorithmic choice at runtime.

Some of these potential performance improvements include datatype optimization, topology caching, synchronization trees, communication schedules, and zero-copy; some of these have been investigated recently by other researchers [2, 8, 12, 20, 21, 22, 23] and provide a strong foundation for realizing these performance improvements. In [23], persistent version of all-to-all operations using Bruck's algorithm [1] were implemented using derived datatypes

to eliminate explicit data movement. Experimental results show that the persistent versions outperform the non-persistent versions by a factor of 2-3 for small message sizes. The algorithm selection and datatype creation overheads are moved to the corresponding *init* operations and the overhead amortized over multiple calls to the all-to-all function. In [22], efficient schedules are computed locally and message combining algorithms are used to implement isomorphic, sparse collective operations. The persistent version of these algorithms move the computation of the schedules and derived datatype creation overhead to the initialization step and reuse them over repeated collective calls. These optimizations provide significant reduction in the latency for small message sizes.

Chapter 2

Design

In MPI, a persistent operation consists of a one-off planning step, followed by zero or more pairs of starting and completion operations, and finally a one-off step to recover (or free) the resources used. Persistent operations for point-to-point communication have been specified as part of MPI since MPI-1 [13]. However, this semantic has not yet officially been extended to other MPI operations, most relatively: collective communication. The effective addition of this feature requires an examination of several critical design considerations.

2.0.1 Initialization

The key interface concept that persistence adds to existing collective operations is the planning step. In a similar manner to the definition for persistent point-to-point operations, the planning step can be seen as splitting the nonblocking function into separate initialization and start functions. For example, the nonblocking function for a collective reduction, `MPI_IREDUCE`, can be split into the initialization of a persistent collective reduction, `MPI_IREDUCE_INIT`, and a separate function call to start that operation, `MPI_START`.

It is expected that the planning and optimization step for persistent collective operations would be expensive in both execution time and resource usage. For this reason, it could be argued that the initialization functions for persistent collective communication should be 'blocking' rather than 'nonblocking'. However, a blocking planning step does not guarantee that all initialization activities must be complete before the first collective communication operation takes place. Achieving such a guarantee would require a precise definition of which activities must be completed as part of initialization. This would predetermine the implementation choices in a manner that is not consistent with the ethos of MPI.

The initialization functions for persistent collective communication could be defined to be 'non-communicating' to be consistent with the initialization functions for persistent point-to-point communication. Such a restrictive semantic would prevent all non-trivial planning and optimization activities from beginning until the first time the persistent collective communication operation was started. One advantage of this approach is that the optimization work occurs if and only if the operation is actually used. However, forcing this work to be delayed until the first 'start' function call would have undesirable consequences on the execution time and resource usage of the first collective communication.

Defining the initialization functions for persistent collective communication to be 'local' instead of 'non-communicating' is less restrictive. The MPI term 'local' only prevents local execution from being dependent on remote MPI operations or remote MPI state. This would permit some planning and optimization activities that involve communication with other MPI processes to begin immediately and continue concurrently with user code and other MPI function calls (if allowed by the thread support level). However, many advantageous optimizations require knowledge of the function parameters supplied by the user at other MPI processes. This in turn requires that the local initialization function call can be matched with the correct initialization call at other MPI processes, which means it should not be 'local'.

Requiring that all participating MPI processes must call the persistent collective initialization function is a natural extension of the existing requirements for blocking and nonblocking collective operations. It implies that persistent collective operations cannot be 'matched' with nonblocking or blocking collective operations, in the same way that nonblocking and blocking collective operations cannot 'match' with each other. This is a necessary constraint for certain desirable optimizations, such as any that require collective coordination among all participating MPI processes.

Ordering is a more complex design decision. Blocking collective operations must be called in the same order by all MPI processes in the same communicator. This ordering

requirement is extended to nonblocking collective functions but not to the completion of those operations. Thus, if one MPI process calls `MPI_IBCAST` followed by `MPI_IALLREDUCE` using a particular communicator, then all MPI processes in that communicator must call those same nonblocking collective functions in that order and without any additional collective operations using that communicator in between. In the absence of tags (or other operation identifiers), ordering of collective function calls is necessary so that 'matching' calls can be correctly distinguished. Thus, either the persistent collective initialization functions, or the start functions, or both must obey this ordering constraint.

Layering the implementation of persistent collectives on top of the existing semantics defined for nonblocking collectives would require that the 'starts' must be ordered (also discussed in section 2.0.2), which would mean that the initialization functions would not need to be ordered. However, for the optimization phase to begin before the first call to `MPI_START`, it must be possible to correctly 'match' the initialization function calls themselves without reference to any specific instance of the communication operation. This provides a strong motivation to require that the initialization functions must obey the collective ordering rule.

Thus, the best design choice for the semantics of the initialization functions is identical to existing definition for nonblocking collective operations:

- called by all MPI processes in the communicator
- correctly ordered within the communicator
- begins communication with other MPI processes (for the purposes of initialization only – the request is 'inactive' until the operation is started)
- returns 'immediately' with no dependency on the state of other MPI processes
- permits communication to continue concurrently with user code
- is permitted (but not required) to synchronize
- requires an additional MPI function call to guarantee completion

For this reason, we choose to prefix the names of the initialization functions with "`I`," as is the custom for nonblocking communication functions in MPI.

The optimization work begun in the initialization function could be completed:

- explicitly - via MPI_WAIT or other completion functions, mandated to occur before the first call to MPI_START;
- implicitly - probably before the first call to MPI_START, or at least before the first communication completes;
- at any time - before, during, or after the first communication, or even on an ongoing basis in response to system environment changes.

The restriction in the first option is unnecessary and contrary to the existing persistent function call sequence. The last two options are indistinguishable to the user of the API (other than by careful measurement of the time taken by MPI calls or via the MPI_T interface, if the MPI library chooses to expose such information).

2.0.2 Starting

For simplicity and consistency within MPI, persistent collective communication operations would ideally be started using exactly the same mechanisms as for persistent point-to-point communication operations; that is, by calling either the MPI_START function or the MPI_STARTALL function. There is currently no requirement for any specific ordering when starting requests. Separate calls to MPI_START can be made in any order and even concurrently using multiple threads, if permitted by the thread support level. The MPI_STARTALL function is defined to act as if each request were started using MPI_START in some unspecified order. Thus, the natural extension of persistent point-to-point semantics is to permit persistent collective operations to be started in any order.

'Matching' of collective operations must be done once during initialization and is reused for each instance of the actual communication operation. This could be implemented inside an MPI library via tagged collective operations or by duplicating the communicator. The tag could be assigned locally during the initialization function based entirely on its ordering. Alternatively, a nonblocking communicator duplication could occur during initialization.

Storing the unique identity of the operation in the MPI_REQUEST data structure permits correct matching each time the operation is started.

However, the ability to start persistent collective operations with different orderings at different MPI processes cannot be layered on top of current nonblocking collectives. Also, some application use-cases do not require the additional flexibility and could guarantee that persistent operations would always be started in the same order by all MPI processes within a particular communicator, or within all communicators. Imposing or guaranteeing strict ordering for starting persistent operations prevents the non-trivial use of MPI_STARTALL and requires thread synchronization in multi-threaded user code for correct usage of MPI_START.

Both of these semantics could be supported in MPI by specifying an assertion via a new MPI_INFO key. Ordered usage is a subset of non-ordered usage so the default semantic should be non-ordered usage and the INFO key should assert that the user will restrict themselves to ordered usage.

If the default is ordered usage, then the INFO key would be a request for additional functionality, *i.e.* non-ordered usage. If MPI cannot support that, then it would need a mechanism to deny the user's request, which suggests a requested/provided interface similar to the thread support level in the MPI_INIT_THREAD function.

If the default is non-ordered usage, then a compliant MPI library must support both semantics; the INFO key is an assertion that users will restrict themselves to ordered usage. MPI may choose to ignore such information or exploit it. This seems to be more in keeping with the original intent of the INFO key functionality. A model implementation, which leveraged nonblocking collective implementation code, might require users to supply the 'ordered' INFO key assertion or to guarantee correct matching in some other way, such as ensuring that only one persistent collective operation at a time be active for all participating MPI processes.

2.0.3 Completion

Completion of each persistent collective communication that was started with `MPI_START` or `MPI_STARTALL` is accomplished by waiting for the request or by repeatedly testing the request. All of the variants (‘all’, ‘any’, and ‘some’) of waiting and testing in MPI are permissible and have the same meanings and semantics as for other MPI operations.

Some applications may wish to guarantee that all optimization activities have completed without starting and completing the first actual communication operation. This desire could be accommodated by creating the request in the active state during initialization and requiring that it be completed prior to the first call to a start function. Normal usage for persistent requests (based on the existing definition of persistent point-to-point communication) is that the request is created in the inactive state, ready for the first call to a start function. However, no long-running activity—in particular no communication—is permitted during point-to-point initialization, so there is no significant work to complete.

If this additional step were added to the sequence of function calls for persistent collective operations, then consistency would dictate a similar addition to the specification of persistent point-to-point operations. This also suggests that the restriction that prevents communication could be lifted, enabling more aggressive optimization for these operations.

2.0.4 Freeing

The final step in the lifecycle for all `MPI_REQUEST`s is to free their associated resources by calling `MPI_REQUEST_FREE`. For persistent requests, this is only allowed when the request is inactive. It is expected that each persistent collective operation could exclusively reserve a significant quantity of limited system resources. Freeing the persistent request gives the user direct influence over when those resources are released. Initializing too many persistent operations without freeing their resources will eventually exhaust the capacity of the system, which could cause impaired performance or even fatal errors.

It is possible to free a persistent request without starting it. In this case, the response in MPI to `MPI_REQUEST_FREE` may be just to mark the request as ready-to-be-freed. When the optimization activities finish, MPI will immediately recover all the resources associated with the request. Alternatively, MPI may respond by attempting to cancel, curtail, or truncate the ongoing optimization activities - and then recover any remaining resources associated with the request. Note that `MPI_REQUEST_FREE` is local, so none of these actions can block and therefore likely must extend beyond the call to `MPI_REQUEST_FREE`.

Chapter 3

Implementation

The prototype implementation of persistent non-blocking collective operations is referred to herein as LibPNBC. It is a derivative of LibNBC (a Library of Non-Blocking Collectives), originally developed by Hoefler and Lumsdaine [10] as a platform-independent library providing pure non-blocking collective functionality. LibPNBC (a Library of Persistent Non-Blocking Collectives) re-identifies and augments this existing library with the necessary elements needed for persistence. The implementation reflects the design through the division of each collective operation into _INIT and _START functions, with a generic conduit to enforce ordering semantics, and a method for storage and retreival of the communication request object and schedule through memory management.

3.1 Requests and Scheduling

The primary objective of persistence is to minimize the inherent redundancy in iterative parallel operations, where communicators repeatedly pass the same arguments. In non-blocking communication, the request object, described in figure 3.2, is of particular importance because of the potential overhead involved in its initialization. With persistence, rather than reallocating an MPI request handle on each iteration, a single request object is created once and reused for each subsequent invocation. This requires caching or storage mechanisms within any component that attempts to implement persistence, so that information about a request can be retrieved upon subsequent communication operations.

Because multiple communications occur, the nature of collective operations necessitates similar storage capabilities through scheduling [10]. LibPNBC uses LibNBC's definition of a communication schedule as an execution plan consisting of one or more interdependent

rounds, each containing zero or more point-to-point operations. During the initialization phase, as the collective operation's algorithm is selected and executed, each internal, atomic send or receive request is added to the component's internal schedule. When started, these requests are then passed to the Point-to-point Messaging Layer (PML) and handled normally before returning to the collective library, where the schedule progression continues until it is complete. This progression algorithm is held in the component source file pnbc.c, specifically the functions described in Figure 3.1.

Listing 3.1: PNBC_Progress Function

```
int PNBC_Progress(PNBC_Handle *handle) {
    res = ompi_request_test_all(...);
    ...
    if (*delim == 0) { /* last round complete */ }
    handle->row_offset = (delim + 1) - handle->schedule->data;
    res = PNBC_Start_round(handle);
}
```

Listing 3.2: PNBC_Start_round Function

```
static inline int PNBC_Start_round(PNBC_Handle *handle) {
...
res = MCA_PML_CALL(...);
res = PNBC_Progress(handle);
}
```

Figure 3.1: PNBC Schedule Progress Functions (pnbc.c)

The data structure of the internal schedule need not be modified for persistence. However, pure non-blocking collective operations operate under the assumption that all communication will be completed before returning to the MPI layer, relinquishing much of the request information in the process.

A method for storing any corresponding collective scheduling data and associating it with the request must be provided at initialization time, and retrieval must be possible when the communication is started. This can be achieved through internal caching mechanisms

within the component, assigning identifying information to each core request so that the necessary schedule can be retrieved from non-volatile storage upon subsequent communication operations. Alternatively, each request that is assigned to a schedule can be held in memory until it is explicitly freed by the caller, when it is no longer needed. As seen in figure 3.2, the PNBC_Handle, which is a substruct of MPI_Request, includes a PNBC_Schedule object as one of its variables which will remain accessible as long as it is not explicity freed internally, and the MPI_Request is maintained in memory.

Listing 3.3: PNBC_Schedule Data Structure

```
struct PNBC_Schedule {
    opal_object_t super;
    volatile int size;
    volatile int current_round_offset;
    char *data;
};
```

Listing 3.4: MPI_Request Data Structure

```
struct ompi_request_t {
    opal_free_list_item_t super; //Base type
    ompi_request_type_t req_type; //Enum indicating the type of the request
    ompi_status_public_t req_status; //Completion status
    void *req_complete; //Flag indicating wether request has completed
    ...
    ompi_request_complete_fn_t req_complete_cb; //Called when request is MPI completed
    ompi_mpi_object_t req_mpi_object; //Pointer to MPI object that created this request
};
```

Listing 3.5: PNBC_Handle Data Structure

```
struct ompi_coll_libpnbc_request_t {
    ompi_request_t super;
    MPI_Comm comm;
    long row_offset;
    int tag;
    volatile int req_count;
    ompi_request_t **req_array;
    PNBC_Comminfo *comminfo;
    PNBC_Schedule *schedule;
    void *tmpbuf;
};
typedef ompi_coll_libpnbc_request_t PNBC_Handle;
```

Figure 3.2: PNBC Relevant Data Structures

3.2 Initialization

The first step in a persistent request is operation initialization. This step is implemented by augmenting each individual collective operation within the component to include an initialization API function of the form MPI_<coll>_INIT e.g., MPI_IBCAST_INIT. The code for these functions is held in each operation's corresponding pnbc_<coll>.c source file within the component subdirectory, as described in 3.9. These functions prepare the operations for use, but stop short of actually starting the operation. This preparation involves the initialization of the MPI_Request (pnbc_handle) object, and the selection and assignment of an the collective schedule to the handle. In some cases, multiple algorithms may be present for the calculation of the communication schedule to be used, which are typically selected based on the communicator size and used for optimization.

To ensure the persistence of the MPI_Request and the corresponding schedule, a method for schedule caching must be provided by each collective operation's initialization function. LibPNBC must be able to retrieve a request's corresponding schedule when it is started. LibNBC itself provides a partial, experimental approach to schedule caching using height-balanced trees to provide efficient lookups, even for a large number of requests. LibPNBC uses a more simplistic approach of maintaining each request in memory until it is explicitly freed by the caller.

The implemented initialization functions also include a placeholder for a MPI_Info flag, as dictated by the design described in chapter two. This parameter is currently unused, but could be used in future implementations to determine the ordering semantics and potential optimizations for certain use cases.

Listing 3.6: Pseudocode: MPI_<coll>_init

```
int ompi_coll_libpnbc_<coll>_init(..., MPI_Info info, ompi_request_t ** request, ...) {
    PNBC_Schedule *schedule; // initalize schedule
    PNBC_Handle *handle; // initialize request handle
    <coll>_sched_<algorithm>(..., schedule, ...); // assign send, receives
    PNBC_Init_handle (comm, &handle, ...); // assign default values to handle
    handle->schedule = schedule; // assign schedule to the handle
    *request = (ompi_request_t *) handle; // reassign the passed request to handle
    return;
}
```

3.3 Starting

In the same manner as traditional point-to-point persistent operations, persistent collective operations rely on a call to a start function, i.e. MPI_START to initiate the communication that was prepared by the initialization function. Implementing this in the LibPNBC component required the addition of a new function to the existing component code, as well as subsequent additions at higher levels in the MCA to make it visible to the MPI API. To avoid challenges associated with mapping MPI_START, our early implementation publishes a temporary function, MPIX_Start for use at the MPI layer. Future implementations will need to provide the necessary bindings for MPI_START when the MPI layer selects the LibPNBC component at runtime.

Listing 3.7: Pseudocode: MPIX_Start

```
static const char FUNC_NAME[] = "MPIX_Start";
int MPIX_Start(MPI_Comm comm, MPI_Request *request) {
    ...
    switch((*request)->req_type) {
        ...
        default:
            OPAL_CR_ENTER_LIBRARY();
            // Invoke the coll component to perform the back-end operation
            err = comm->c_coll->coll_libpnbc_start(request);
            OPAL_CR_EXIT_LIBRARY();
    }
}
```

Listing 3.8: Pseudocode: coll_libpnbc_start

```
int ompi_coll_libpnbc_start(ompi_request_t ** request) {
    PNBC_Handle *handle;
    PNBC_Schedule *schedule;
    // assign handle to passed request, still pointing to
    // the initialized handle from MPI_<coll>_init
    handle = (PNBC_Handle *) *request;
    schedule = handle->schedule; // assign schedule from handle
    PNBC_Start_internal(handle, schedule); // begin schedule execution
    return;
}
```

Figure 3.3: MPIX_Start Functions

LibPNBC currently provides the function mappings and MPI layer hooks necessary to fire both initialization and start functions required for persistence. The addition of the initialization and start functions, along with the memory maintenance of collective schedules, into the existing LibNBC collective operations enables the base functionality of persistent non-blocking collectives. This functionality is expected to provide a modest boost to inter-process communication for large-scale use of collective operations. As LibNBC is used by other MPI implementations, we view this model implementation and prototyping effort as reusable beyond Open MPI, though with modifications, and much room for optimization.

Listing 3.9: Blocking Send

```
for (i=0; $i<$max; i++) {
   compute(bufA);
   MPI_Bcast(bufA,...,rowcomm);
   compute(bufB);
   MPI_Reduce(bufB,...,colcomm);
}
```

Listing 3.10: Non-Blocking Send

```
for (i=0; $i<$max; i++) {
   compute(bufA);
   MPI_Ibcast(bufA,...,rowcomm, &req[0]);
   compute(bufB);
   MPI_Ireduce(bufB,...,colcomm, &req[1]);
   MPI_Waitall(2, req, ...);
}
```

Listing 3.11: Persistent Send

```
MPI_Ibcast_init(..., &req[0]);
MPI_Ireduce_init(..., &req[1]);
for (i=0; $i<$max; i++) {
   compute(bufA);
   MPIX_Start(req[0]);
   compute(bufB);
   MPIX_Start(req[1]);
   MPI_Waitall(2, req, ...);
}
MPI_Request_free(&req[0]);
MPI_Request_free(&req[1]);
```

Figure 3.4: Sample MPI Program Code: Non-Blocking Send vs. Persistent Send

3.4 Development Methods and Tools

3.4.1 OpenMPI

As a widely-deployed, extensible, and well documented implementation of MPI, OpenMPI has been chosen as the codebase on which to implement this initial prototype of persistent non-blocking collectives. As of June 2017, the main development source code repository for OpenMPI is provided publicly at https://github.com in the open-mpi/ompi repository (https://github.com/open-mpi/ompi.git).

Detailed development documentation and other critical information is provided at the same location. Stable releases, publications, user documentation, and other relevant materials are also made available at https://www.open-mpi.org. Additionally, OpenMPI user and developer mailing lists are available for public subscription (https://www.open-mpi.org/community/lists) as an interactive resource.

To summarize:

- Development Source Code: https://github.com/open-mpi/ompi.git
- Development Documentation: https://github.com/open-mpi/ompi/wiki
- Stable Releases and Additional Documentation: https://www.open-mpi.org
- Community Mailing Lists: https://www.open-mpi.org/community/lists

3.4.2 Open MPI Implementation Model

To promote portability, OpenMPI uses a layered implementation model, separating functionality into three abstracted groups. OPAL, the Open Portability Abstraction Layer, defines essential functionality for individual processes to interact with the operating system, with some degree of platform independence. ORTE, the Open Runtime Environment provides a runtime system in which to perform process management and platform-specific API interactions, i.e. interaction with workload managers. OMPI provides the interface syntax and semantics necessary to initiate MPI functionality from the user application [17].

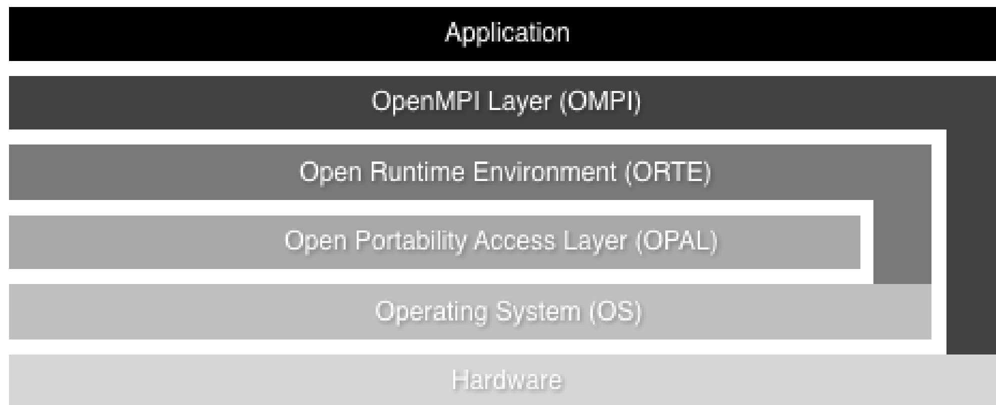


Figure 3.5: OpenMPI Abstraction Model[17]

This model is reflected in the OpenMPI project directory structure and nomenclature, and serves as a good reference for interpreting STDERR and STDOUT messages at compile and run time.

3.4.3 The Modular Component Architecture

In OpenMPI, within each abstraction layer exists a file architecture known as the Modular Component Architecture (MCA), where additional functionaility can be added. The MCA provides extensibility through the use of compiler toolchains, file nomenclature, syntax, and protocol that guides build automation.

In broad terms, the MCA defines frameworks, and more narrowly, components c.f. Figure 3.6. Frameworks are generalized groups of similar operations, such as collective (coll) or point-to-point (pml). A component is a set of specific functional implementations, within a framework, consisting of algorithmically (or otherwise) differentiated code.

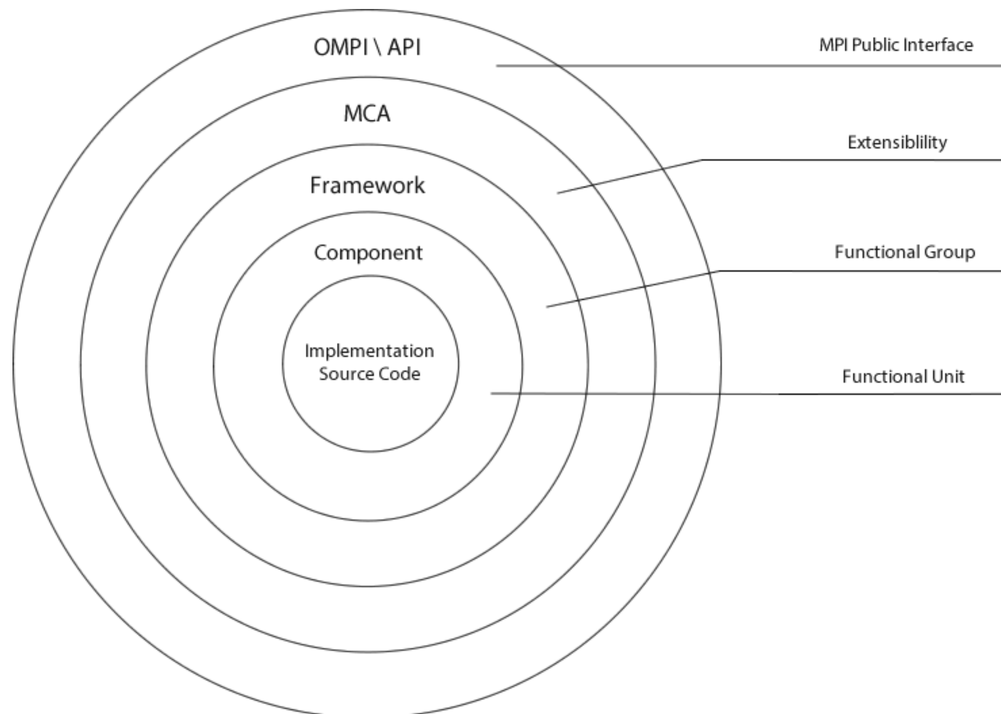


Figure 3.6: Modular Component Architecture Conceptual Model

Nearly all of OpenMPIs core functionality, including point-to-point communication (PML) and data transfer (BTL) is implemented in the MCA. Adding custom component-level functionality involves following a loose protocol described in the OpenMPI documentation, currently held at https://github.com/open-mpi/ompi/wiki, and in the framework source code comments. Generally, it consists of the creation of a specific directory structure, the developer's choice of one of three standard makefile formats, and corresponding source code with accompanying framework-specific semantics or syntax.

```
<trunk>
    ↳ autogen.pl // automated build, initialization
    ↳ mpi/<source_language>// api-level functions
    ↳ ompi
        ↳ mca/<framework>
            ↳ ./base // minimal, global framework functionality
            ↳ ./<component>
                ↳ configure.m4, configure.in, or makefile.am
                ↳ [framework]_<component>_component.h // required interface
                ↳ <framework>_<component>_component.c // required definitions
                ↳ source code
    ↳ opal
        ↳ . . .
    ↳ orte
        ↳ . . .
```

Figure 3.7: OpenMPI Source: Locations of Interest

When properly located and defined, the build process for OpenMPI will crawl the MCA's frameworks looking for components and then incorporate the necessary compiler directives into a standard configure script. This search and include process is performed using a GNU Autotools based Perl script, autogen.pl, located at the root of the OpenMPI source.

By default, OpenMPI makes implicit decisions at runtime (via mpirun) on which components are selected and enabled for use based on hard-coded priority levels and functionality requested by the application. This selection process can be overridden by providing mpirun with specific parameters specifying the desired component selection. This can be useful in

debugging as a more deterministic way of selecting the desired code to be executed.

```
$ mpirun --mca coll libpnbc,libnbc,basic,self -np 16 ./mpi_program
```

Figure 3.8: Forcing Specific Component Selection at Runtime

3.4.4 Component Development

Figure 3.9 describes the structure of the LibPNBC MCA component. Each component must provide a corresponding definition of itself within the framework interface which is used to incorporate and publish any public functions for use at the API level. Much of this identification is coded in the framework's main interface .h file, in the case of LibPNBC, coll.h. The collectives interface defines two primary structs: one for association with the MPI communicator, and one for association with the module (running instance of a component).

<trunk>/ompi/mca/coll
  ↳ coll.h - collective framework main interface
  ↳ libpnbc - libpnbc component subdirectory
    ↳ coll_libpnbc.h - libpnbc main header
    ↳ coll_libpnbc_component.c - libpnbc component functions and pointers
    ↳ Makefile.am - raw component makefile
    ↳ Makefile.in - interpreted component makefile
    ↳ owner.txt - component developer information
    ↳ pnbc.c - global schedule and progress functions
    ↳ pnbc_i<coll>.c - collective operation _init and algorithms
    ↳ pnbc_internal.h - datatypes, structs, and function prototypes
    ↳ pnbc_neighbor_helpers.c - neighborhood collective helper functions
    ↳ pnbc_start.c - code for MPIX_Start

Figure 3.9: LibPNBC Component Source Files

3.4.5 Source Control

The source control system used for OpenMPI is git, with the source code published via github.com in the open-mpi/ompi repository. In order to prepare for potential inclusion of any new features into the main OpenMPI distribution, a user fork is created using github.com's fork utility. A new branch is then created, from which new code modifications are made, and then potentially introduced into the main OpenMPI development repository with a pull request. Once the forked development repository and branch are created, a number of software tools and command line utilities can be used to effectively program and debug the source code.

3.4.6 Software Tools

The prototype developed in this research was performed primarily using a Linux operating system with standard open-source tools for coding, debugging, and benchmarking. In addition to a good terminal and text editor, some other recommended tools are:

- Linux GCC Toolchain
  - gcc 4.9.3
  - automake>=1.13
  - autoconf>=2.69
  - m4>1.4.16)
- Eclipse Parallel Tools Platform (CPP or PTP)
  - GNU Debugger (GDB)
  - GDB-MI (GDB Machine Interface Protocol)
  - Session Debug Manager (SDM)
- Valgrind 3.11.0

3.4.7 LibNBC

The implementation of persistent non-blocking collectives as demonstrated here, borrows heavily from OpenMPIs Modular Component Architecture (MCA), in particular the LibNBC component. LibNBC is currently distributed as part of the OpenMPI release, as well as in other MPI implementations, and also as a standalone library. As is LibPNBC, the original LibNBC component code as implemented in OpenMPI, is located within the MCA directory structure. Detailed information is available from the LibNBC development website, hosted at https://htor.inf.ethz.ch/research/nbcoll/libnbc.

```
$ cd <trunk>/ompi/ompi/mca/coll
$ cp -r libnbc/ libpnbc
$ cd libpnbc
$ rename nbc pnbc *nbc*
$ sed -i 's/nbc/pnbc/g' *.c Makefile.*
$ cd <trunk>
$ ./autogen.pl
```

Figure 3.10: Initial Component Re-identification of LibNBC to LibPNBC

3.4.8 Build Process

After obtaining the source code, the autogen process typically needs to be run before the first build attempt on a system, and when any significant changes to the MCA are made, such as adding or removing components. Upon successful completion of autogen.pl, OpenMPI code can be built using the typical configure, make, make install pattern[17].

As different functionality is tested, it is convenient to be able to switch between different development versions of OpenMPI. The typical Linux environment variables for controlling this are described in 3.12 in modulefile (tcl) format, which can be translated to export commands for use in bash shell scripting.

```
$ git clone openmpi_source
$ cd openmpi_source
$ ./autogen.pl
$ mkdir ../build
$ cd ../build
$ ../openmpi_source/configure --prefix=/custom/path --with-devel-headers
$ make
$ make install
```

Figure 3.11: General OpenMPI Build Pattern

```
set             root              /my/custom/openmpi/install/location

prepend-path    PATH              $root/bin
prepend-path    MANPATH           $root/share/man/
setenv          MPI_HOME          $root
setenv          MPI_RUN           $root/bin/mpirun

prepend-path    LD_RUN_PATH       $root/lib
prepend-path    LD_LIBRARY_PATH   $root/lib
prepend-path    CPATH             $root/include
```

Figure 3.12: Environment Settings for Custom OpenMPI Installation

Chapter 4

Results and Discussions

4.1 Benchmarking

A suitable starting point for performance evaluation and benchmarking of LibPNBC is NBCBench, the corresponding microbenchmark used initially to evaluate LibNBC. The use of NBCBench to measure the impact of persistence on collective operations leverages proven benchmarking methods [11], and allows for direct comparison and validation of results with prior benchmark tests. This fits well with the main objectives of this research, which are to measure overhead, determine if there is any initial performance benefit from the persistence API itself, and to identify any potential opportunities for optimization.

NBCBench uses high precision timers to measure the execution time of a collective operation. For a blocking collective, this is simply the difference between the end time and start time of the operation. For a nonblocking collective, the operation time is measured in the call to the operation as well as the additional call to `MPI_Wait`. For persistent nonblocking collectives the measurement includes the operation initialization (`MPI_<coll>_INIT`), start (`MPIX_START`) and completion (`MPI_WAIT`). Each rank within a communicator performs these measurements, which are ultimately aggregated and the median value is presented as output [11]. The specifics of timing the persistent collective operations are shown in detail in Figure 4.1 and Figure 4.2.

As described in Figure 4.2, `MPI_Wtime` is issued at several different points in PNBCBench to measure each step in the operation call pattern. The call to `MPI_Barrier` before each iteration is issued to prevent any significant process skew. Computational overlap is synthesized in the call to `docompute_test`, which issues a number of `MPI_Test` calls in a time based loop to simulate a independent computational workload. The benchmark program accepts

```
MPI_Barrier(comm);
start_time = MPI_Wtime();
MPI_Ibcast(bufA,...,&req);
issue_time = MPI_Wtime();
test_time = docompute_test();
before_wait = MPI_Wtime();
MPI_Wait(comm);
after_wait = MPI_Wtime();
total_time = after_wait - start_time;
```

Figure 4.1: Pseudocode for NBCBench time measurements

```
MPI_Barrier(comm);
start_time = MPI_Wtime();
if(needs_init) {
  MPIX_Ialltoall_init(bufA, ..., info, &preq);
}
MPIX_Start(mpiargs->comm, &preq);
issue_time = MPI_Wtime();
test_time = docompute_test();
before_wait = MPI_Wtime();
MPI_Wait(comm);
after_wait = MPI_Wtime();
total_time = after_wait - start_time;
```

Figure 4.2: Pseudocode for PNBCBench time measurements

a collective operation and runs a user-defined number of trials in the exponential series 2^n with n within the range $pmin$ to $pmax$, also user-defined.

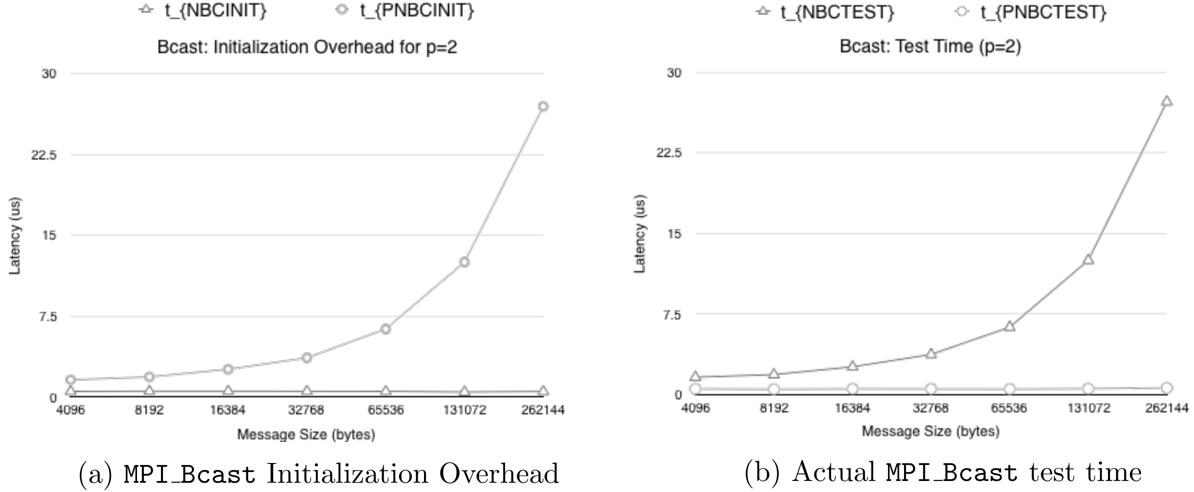
The NBCBench and PNBCBench benchmark testing in this research was performed using the Auburn University Hopper Cluster, a machine employing Lenovo System x3550 M5 servers with 2xE5-2660 Intel Haswell v3 CPUs at 2.60 GHz. Trials were performed from 2 to 512 MPI processes over a range of 4,096 to 262,144 byte data sizes. Each time measurement using `MPI_Wtime()` is repeated ten times and a reduction operation is performed for each rank. NBCBench can perform a minimum, maximum, median, or average calculation for this operation [19]. We have chosen median (the default) as the final time output for all ranks being shown as the final time calculation.

4.2 Overhead

The expected benefit of adding persistent requests is increased overall efficiency for repeatedly used operations, where processes pass the same arguments across the group of processes sharing a communicator many times in an iterative computation. The reuse of requests in memory shortens the critical path of MPI nonblocking collective operation, and thus should result in a shorter overall execution time. This was observed for the majority of operations tested with the described benchmarking method.

Of particular interest is the amount of overhead, if any, that is associated with the initialization phase and the behavior of the operations after the `MPIX_START` is issued. Because the code for the LibPNBC `MPIX_I<coll>_INIT` functions is essentially the same as that of the corresponding `MPI_I<coll>` functions in LibNBC (with the exception of the immediate execution of the collective schedule) it is expected that initialization overhead would be minimal. Given the absence of this granularity in LibNBC, there is no direct comparison available. Thus, as seen in figure 3, the initialization phase is actually measured for both the `MPIX_<coll>_INIT` and `MPIX_START` and compared to the timing of `MPI_I<coll>`.

For most operations, an increase in latency was observed for LibPNBC in the initialization phase (*cf*, Figure 4.3a for `MPIX_Bcast_init`). This is likely because of the difference in progress advancement between the two components. Because LibNBC processes requests more dynamically, progress does not occur on the first schedule round for that component, delaying the initial progress until the issuance of `MPI_TEST`. LibPNBC does attempt progress in the initial schedule round. For PNBC, this choice trades off latency against overlap; early progress on the new operation encourages low latency, whereas delegating progress to the Open MPI framework permits greater overlap. This is further supported by the results of `MPI_TEST` latency for most operations (*cf,* Figure 4.3b for the actual broadcast operation).



(a) `MPI_Bcast` Initialization Overhead

(b) Actual `MPI_Bcast` test time

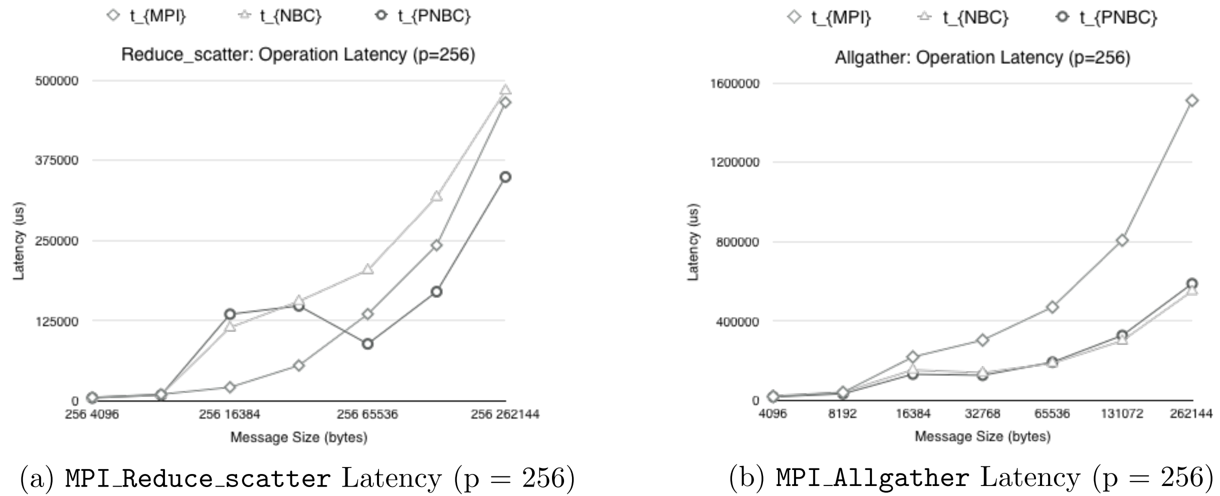
Figure 4.3: Initialization Overhead and Test Progress

4.3 Crossover

As the communicator size scales out, it is expected that the performance impact of persistence will be more prevalent because of the inherent efficiency in memory allocation and reference. A performance benefit should also be apparent with increasing message sizes, as buffers remain relatively static for persistent requests within each iteration, allowing for machine and network architectures to leverage optimization techniques such as caching and pipelining. Additionally, for persistent collective operations, there is no need to duplicate the instructional work of communication schedule creation within iterative computations.

As expected, in the majority of tested operations, a slight decrease in latency was observed for LibPNBC over LibNBC in both communicator and message size. This is most readily seen in Figure 4.4a for `MPI_Reduce_scatter`, where a significant speedup was observed over LibNBC, particularly for larger message sizes in the p=256 communicator. Potentially, one contributing factor to this is the relatively significant amount of overhead incurred in the schedule creation for `MPI_Reduce_scatter`, which is a multiple round operation, consisting of send, receive, operation, and barrier operations in higher frequency than all other measured collectives. Conversely LibPNBC underperformed LibNBC for the `MPI_Allgather` and `MPI_Alltoall` operations, with slightly higher latencies as shown in Figure 4.4b. Further

evaluation of the communication patterns of these outlying operations may provide additional insight into further optimization for persistence.



(a) `MPI_Reduce_scatter` Latency (p = 256) (b) `MPI_Allgather` Latency (p = 256)

Figure 4.4: Operation Latency Outliers

Given that this first attempt at implementation of persistent nonblocking collectives has not yet fully explored opportunities for optimization, these results indicate a potentially impactful new communication method for MPI.

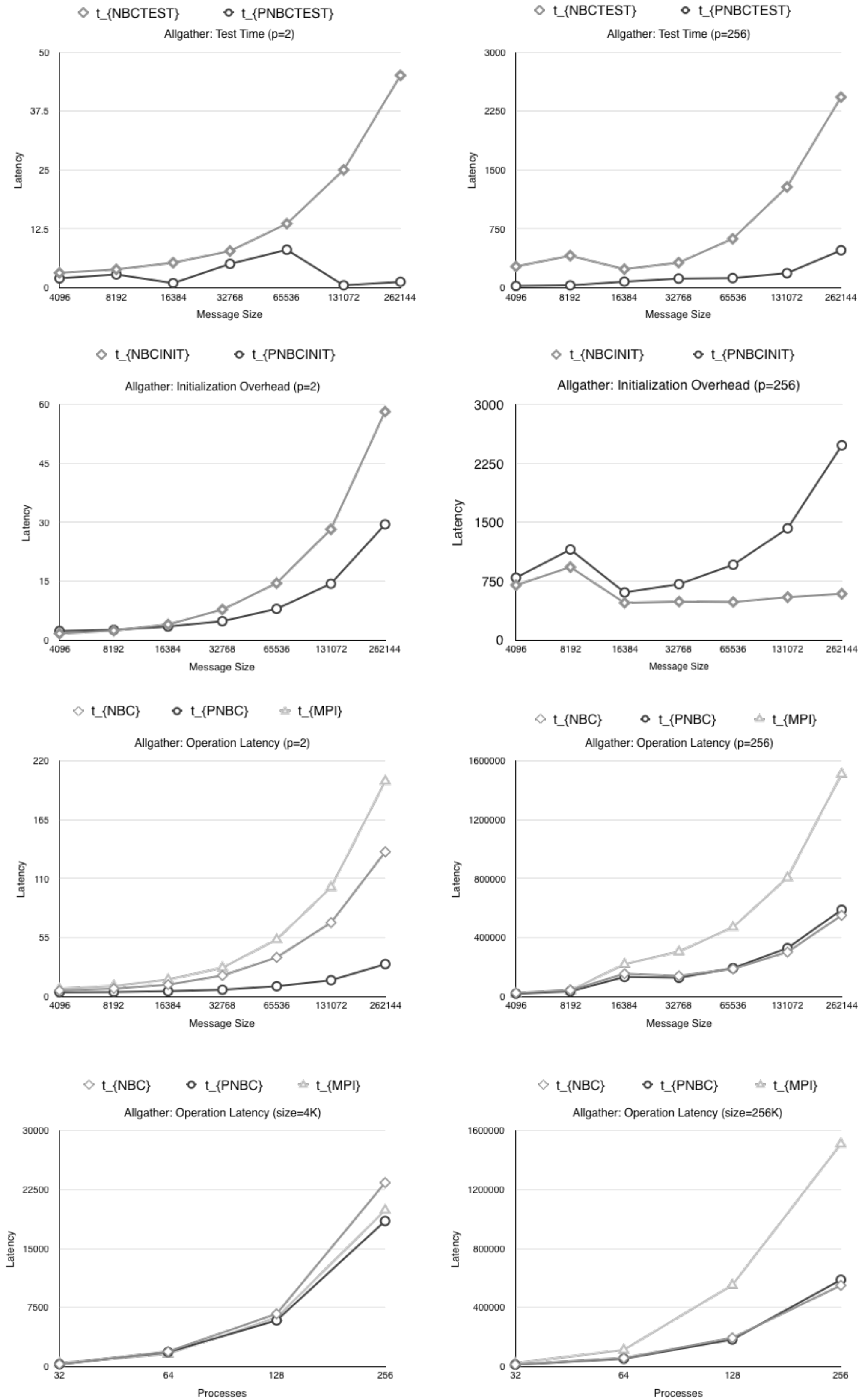


Figure 4.5: Allgather Operation Benchmark Results

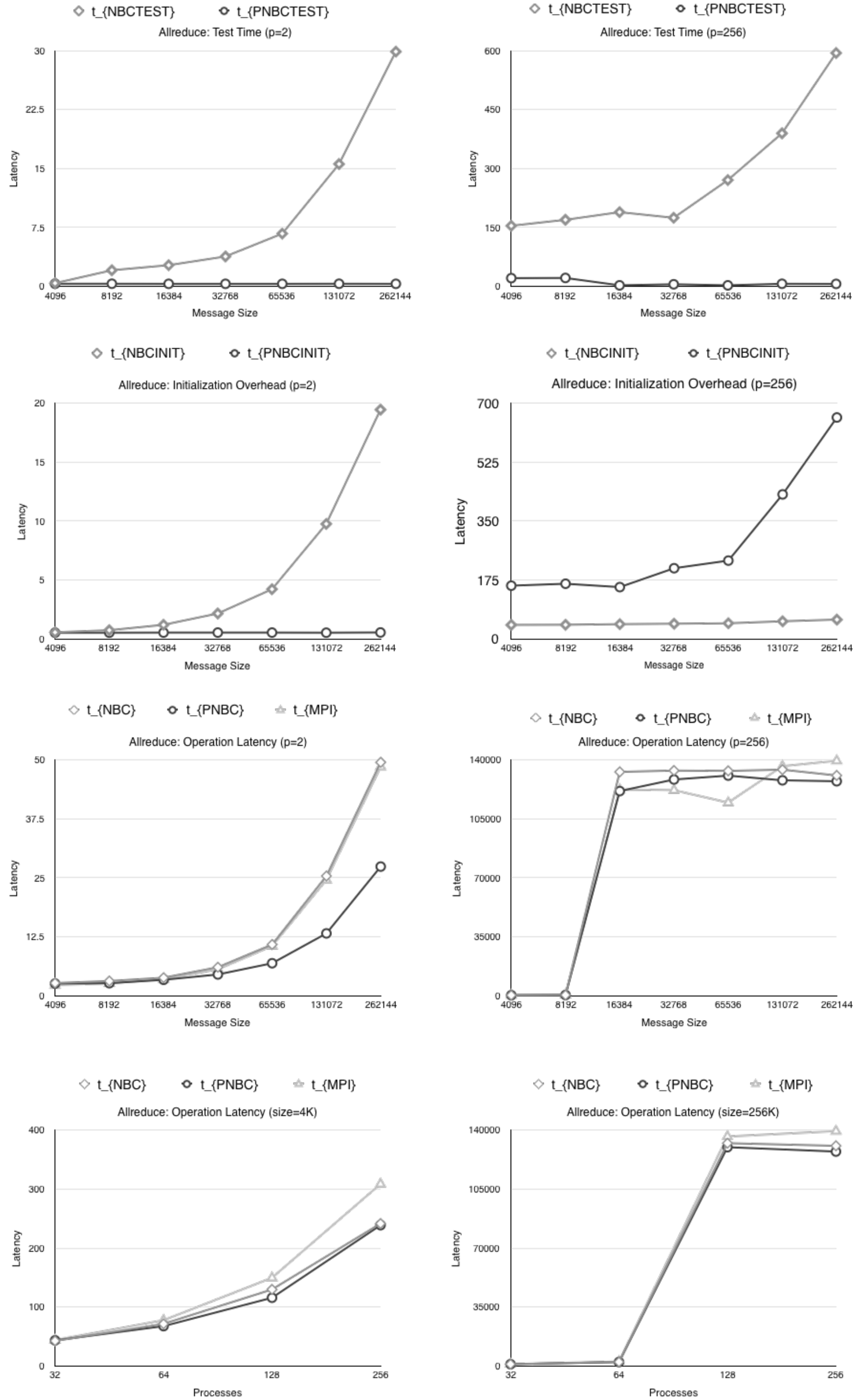


Figure 4.6: Allreduce Operation Benchmark Results

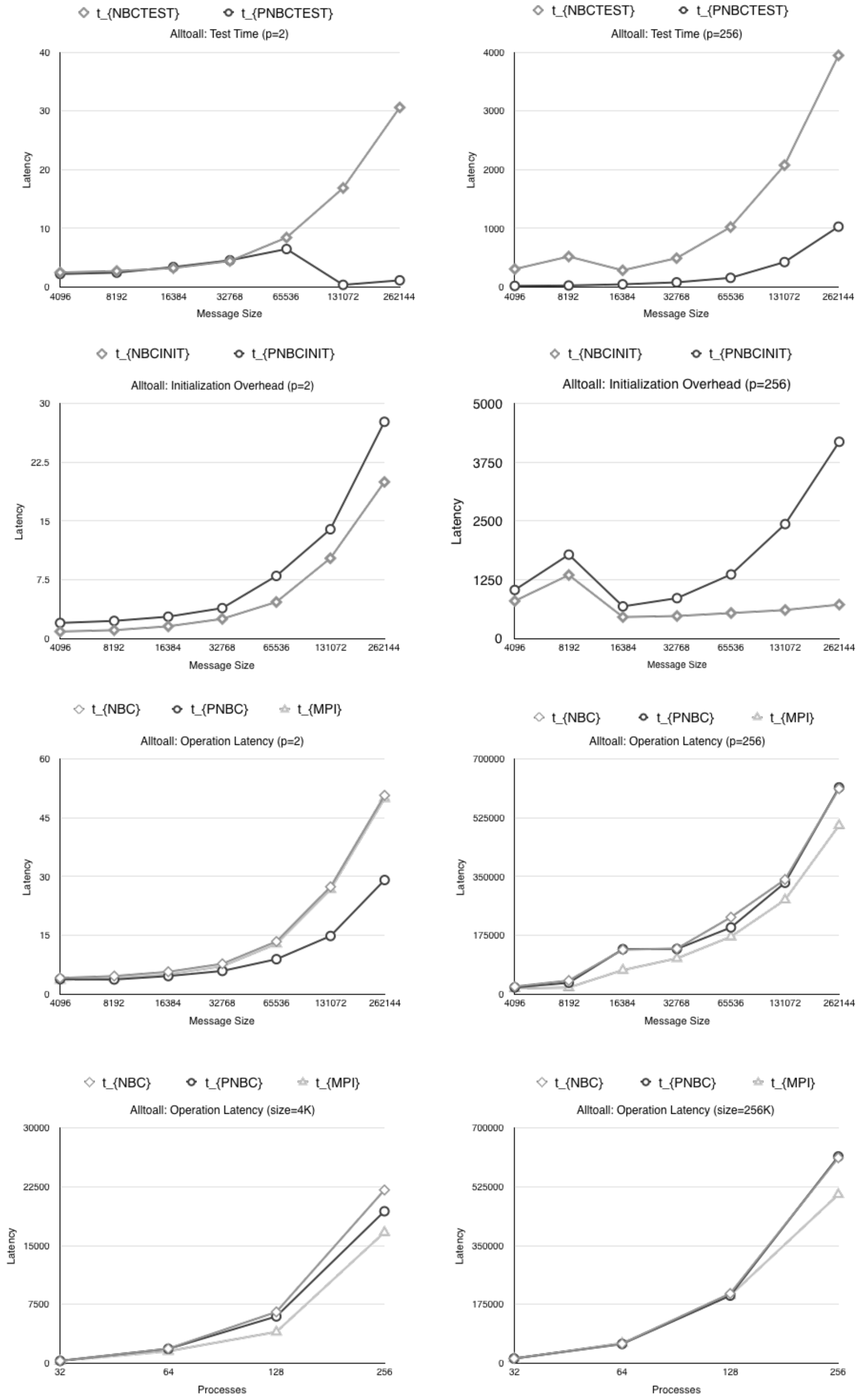


Figure 4.7: Alltoall Operation Benchmark Results

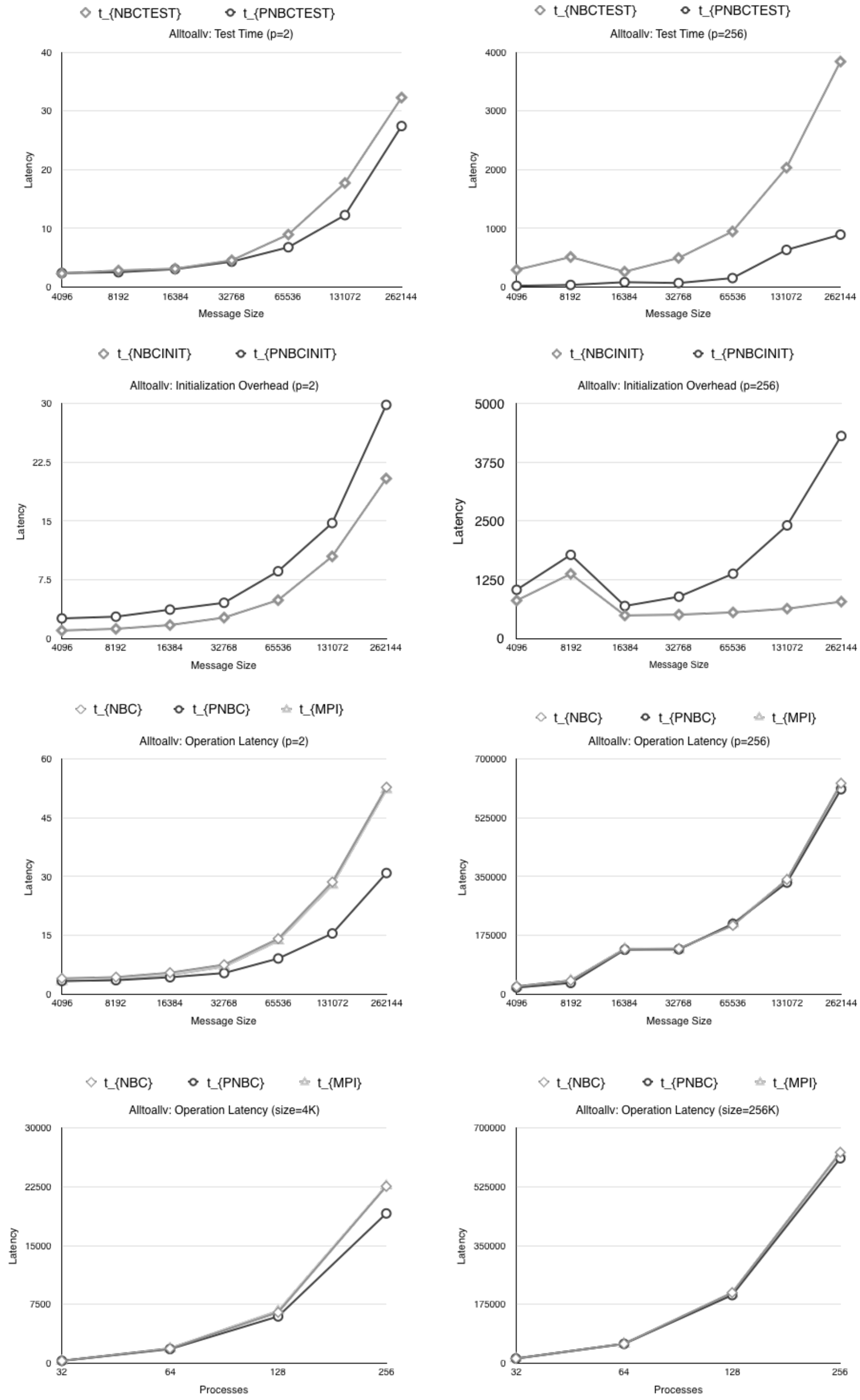


Figure 4.8: Alltoallv Operation Benchmark Results

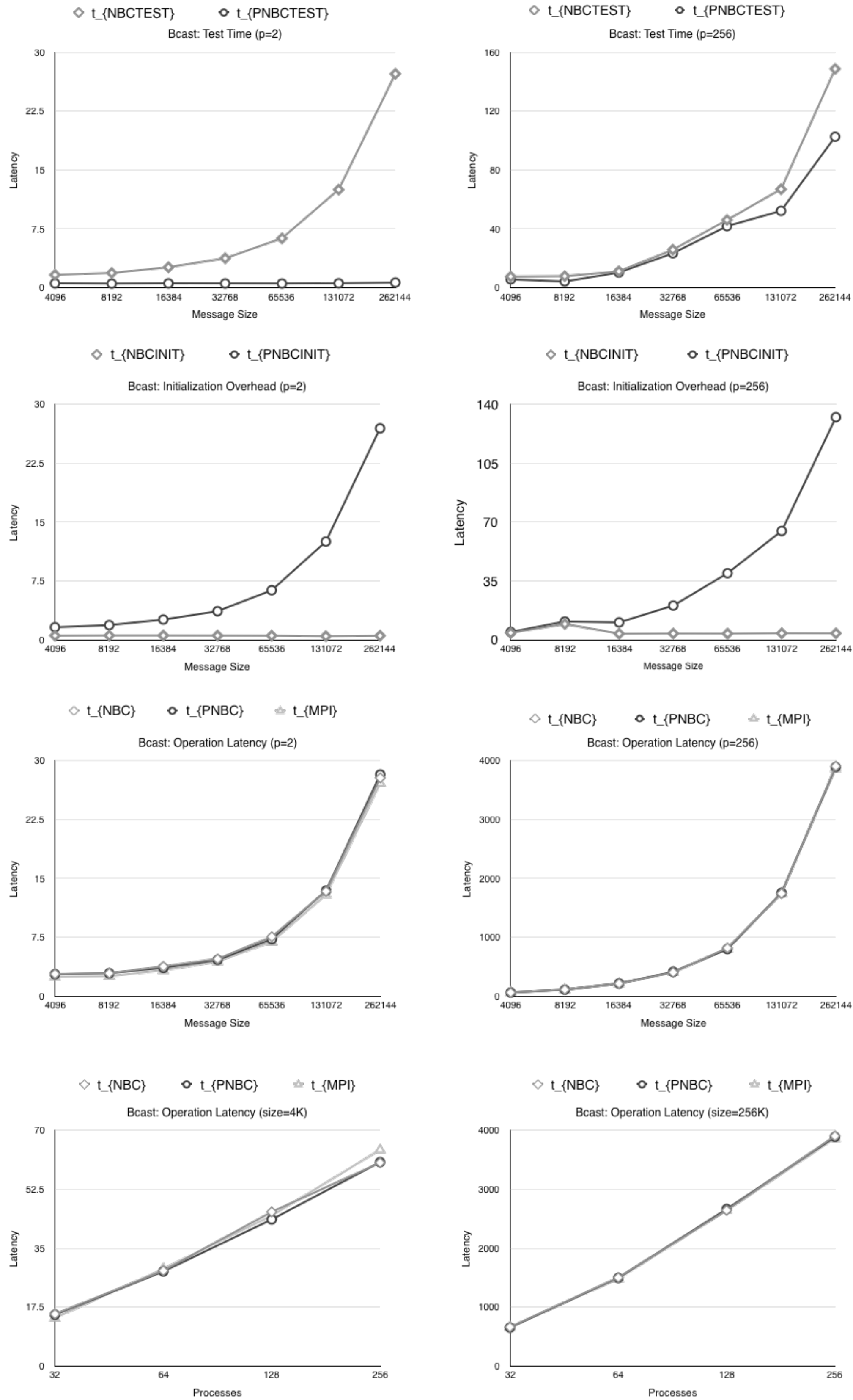


Figure 4.9: Bcast Operation Benchmark Results

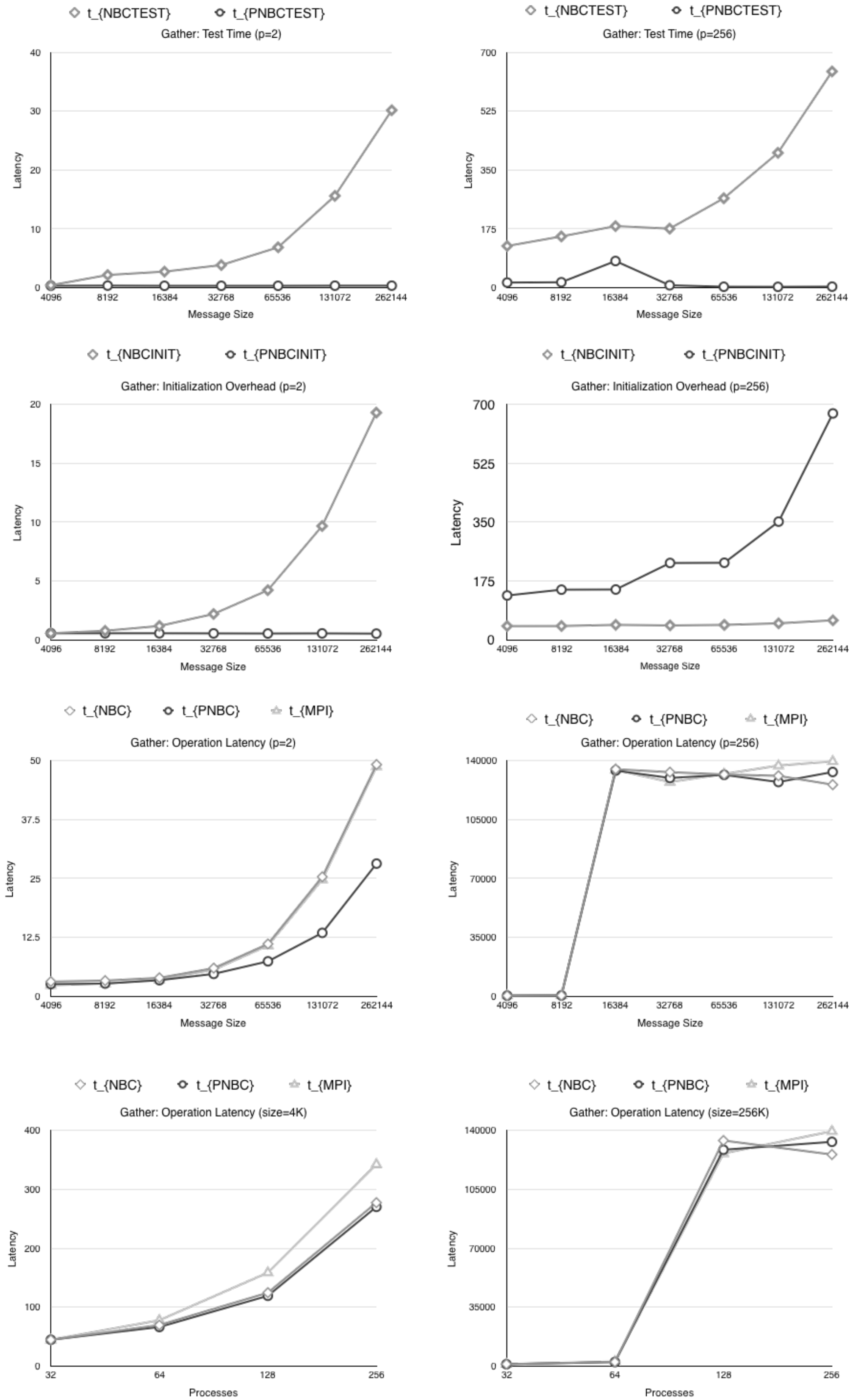


Figure 4.10: Gather Operation Benchmark Results

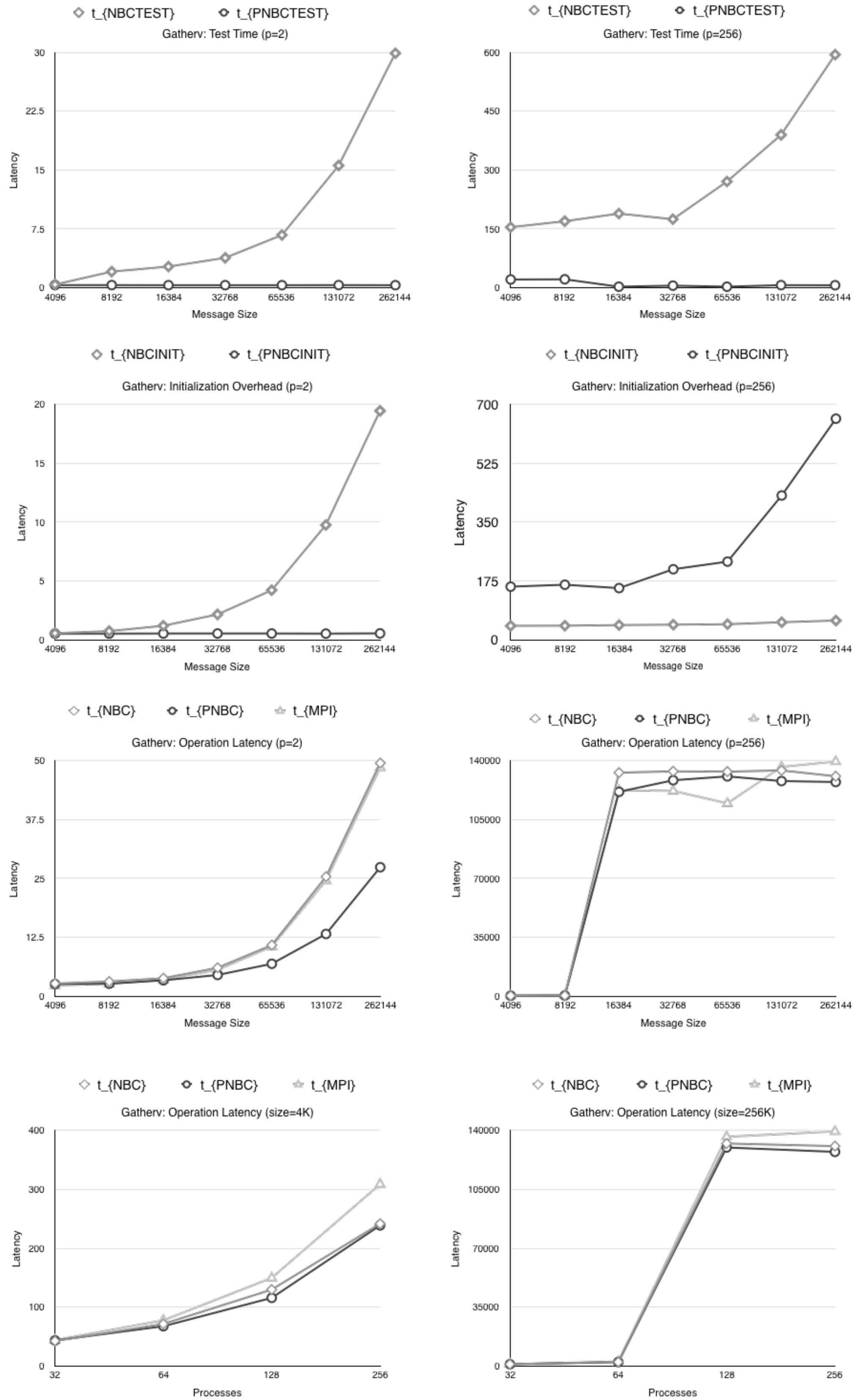


Figure 4.11: Gatherv Operation Benchmark Results

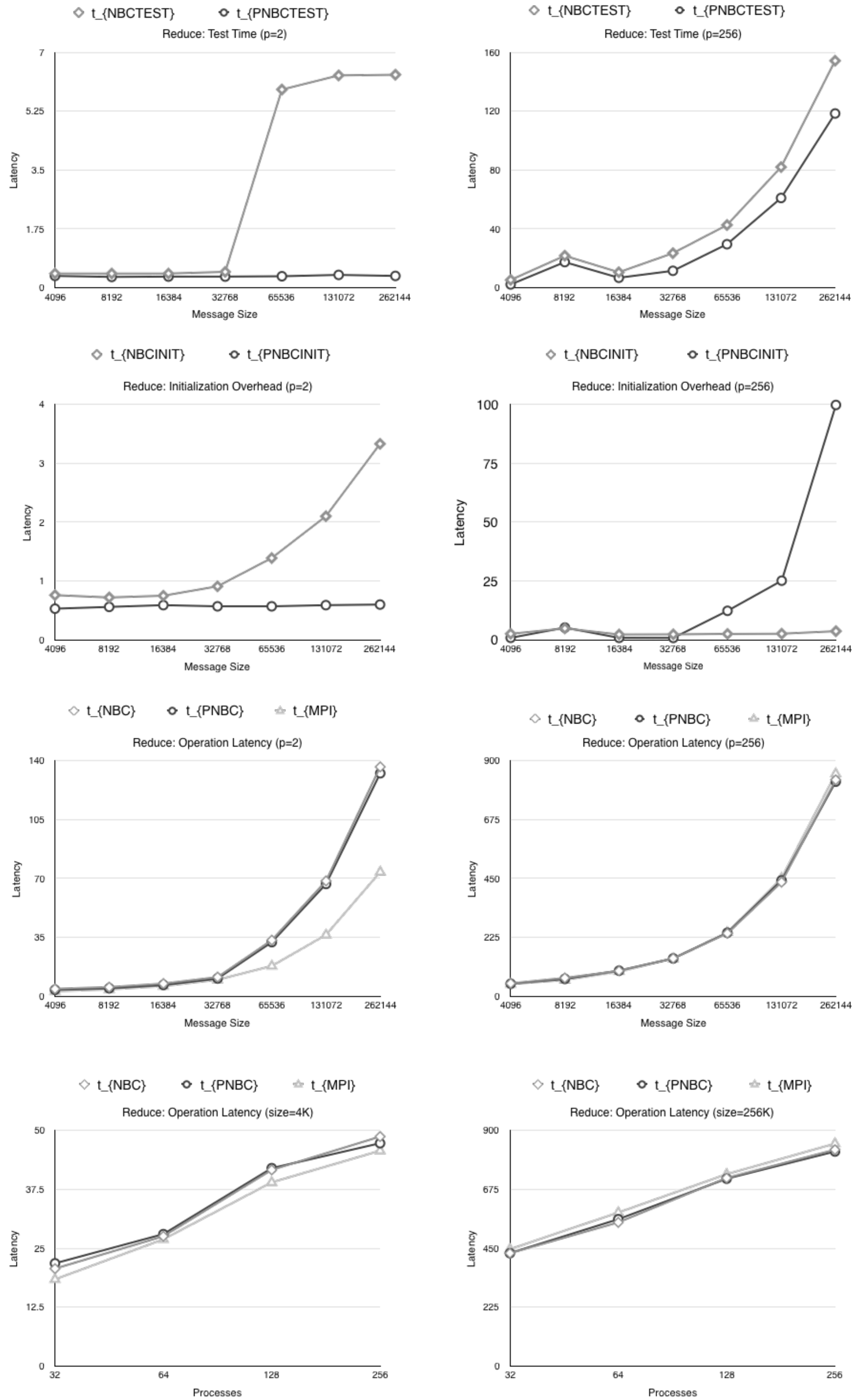


Figure 4.12: Reduce Operation Benchmark Results

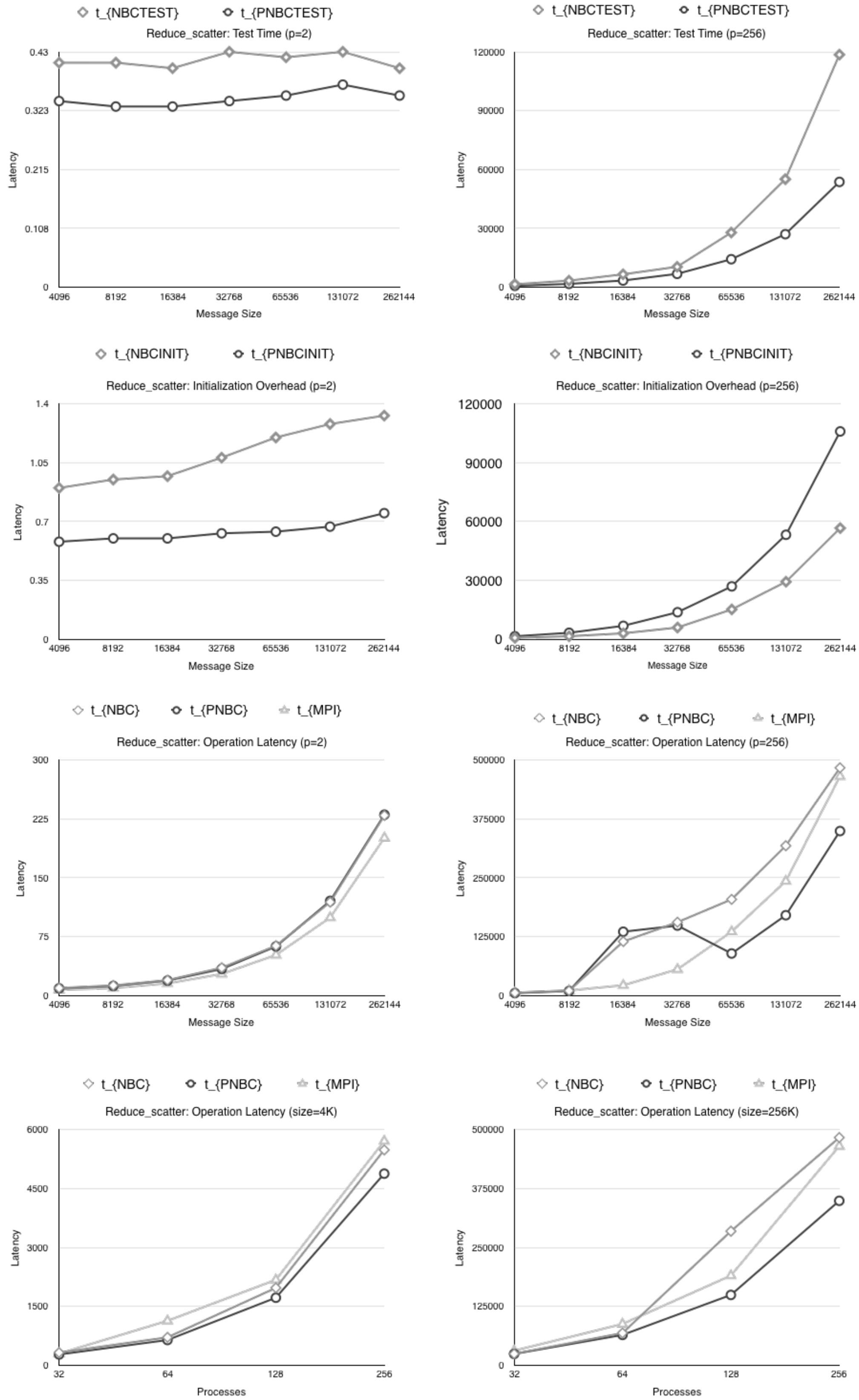


Figure 4.13: Reduce_scatter Operation Benchmark Results

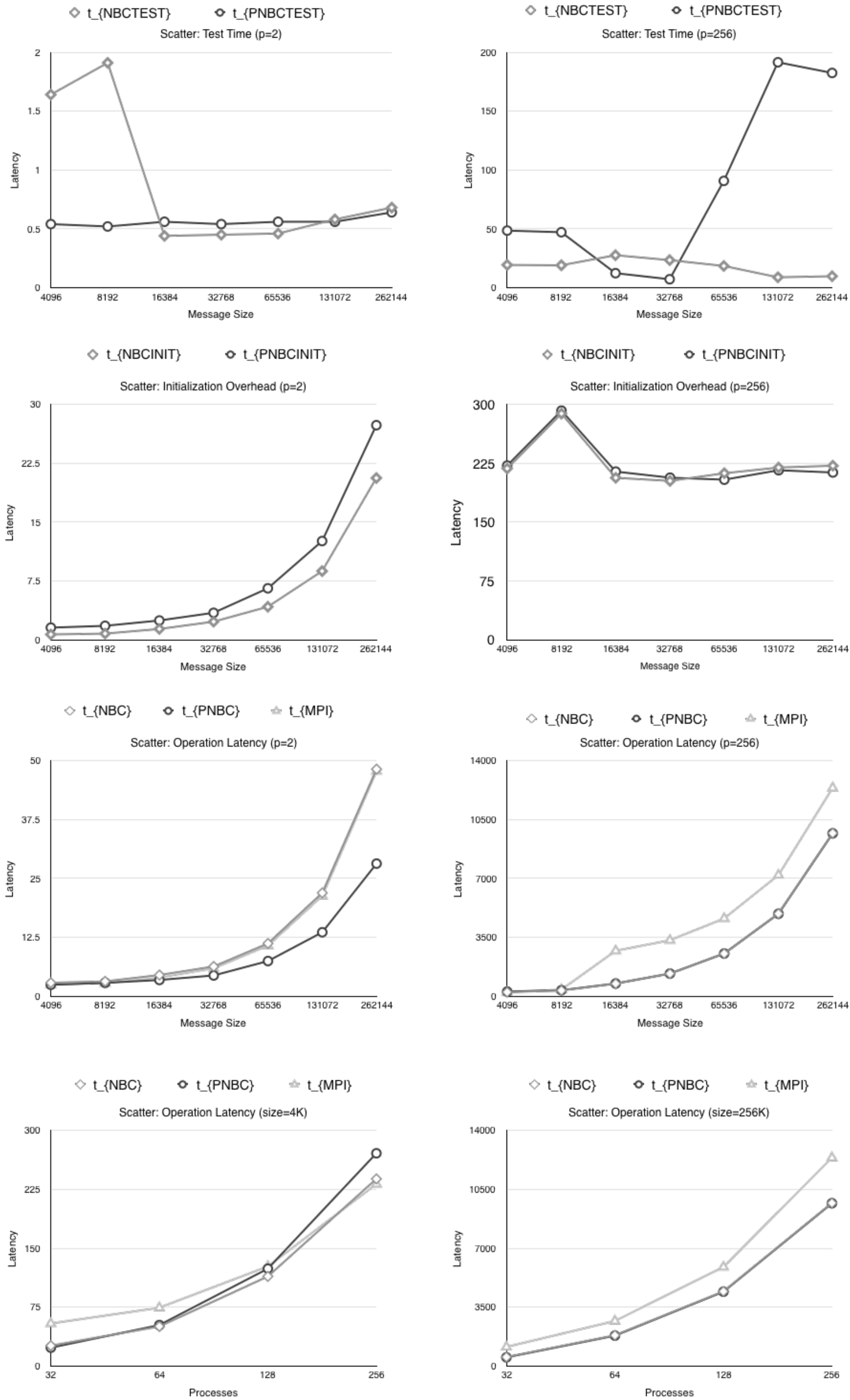


Figure 4.14: Scatter Operation Benchmark Results

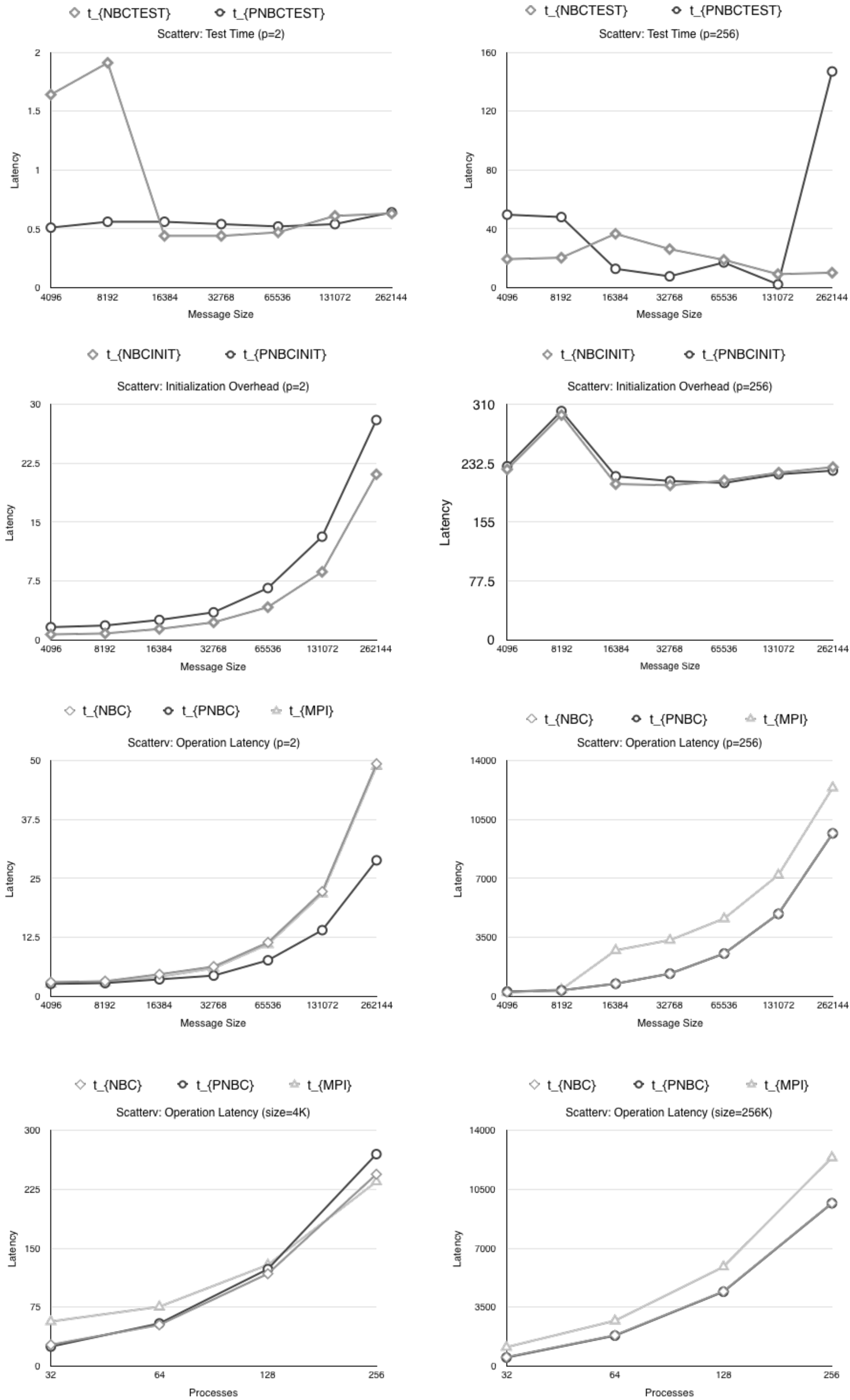


Figure 4.15: Scatterv Operation Benchmark Results

Chapter 5

Future Work

The main future work following on from this prototype implementation of persistent collective communication will be algorithmic improvements; that is, choosing appropriate algorithms from a potentially wider optimization space and implementing the necessary heuristics, operation schedule, and caching mechanisms. Some of these algorithms are discussed in Section 1.3. However, some additional work on the infrastructure code itself would be beneficial both in terms of software engineering and performance optimization.

5.1 Point-to-Point Persistence

As described in 3.1, the core SEND and RECV operations executed internally by the LibPNBC component are passed as non-blocking requests to the point-to-point layer. Converting these core requests to persistent operations would likely provide further efficiency. This functionality is held primarily in PNBC_Start_round function within the pnbc.c source file, where calls are issued to MCA_PML_CALL() using the corresponding non-blocking SEND or RECV operation.

5.2 Non-Algorithmic Optimizations

For both LibPNBC and LibNBC on which it is based, the schedules for all collective operations recreate nonblocking point-to-point communication requests for each round just before executing that round. This means that some of the time taken to perform the communication is actually used to create parts of the schedule. For LibPNBC, this could be improved by creating all rounds once at the beginning (during the initialization function) and using persistent point-to-point communication requests at each round. However, that

requires invasive code-changes to the whole scheduling storage/usage code and is beyond the scope of the current prototype. Layering LibNBC on top of persistent point-to-point instead of nonblocking point-to-point is also possible but it would give a performance benefit if the schedule can be safely cached and reused. However, the memory used by the cached non-blocking schedules cannot easily be recovered until `MPI_FINALIZE` because `MPI_REQUEST_FREE` has no effect on an inactive `MPI_REQUEST` from a nonblocking operation.

Some implementations of point-to-point in Open MPI implement persistent point-to-point using nonblocking point-to-point, re-creating a new nonblocking operation each time `MPI_START` is called. This suggests possible optimization work that could be done in the point-to-point implementation on which both LibNBC and LibPNBC currently rely.

The LibNBC component was originally created as a library external to MPI, one reason that its implementation is layered on top of MPI point-to-point functions. The LibPNBC component has inherited this feature. However, now that these components are fully integrated into the Open MPI framework, other implementation options become available. For example, the PML components (protocol/messaging layer) delegate to BTL components (byte transfer layer) to transfer data; that suggests the collectives could be implemented directly on top of the BTL components rather than via the point-to-point functions. Bypassing the PML components would remove point-to-point matching logic from the critical path of nonblocking and persistent collective communication. Alternatively, collective operations could be implemented by directly using native network Remote Direct Memory Access (RDMA) and Atomic Memory Operations (AMOs) in similarly to recent work that provided an optimized implementation of the MPI single-sided functionality for Open MPI [9]. For sufficiently capable hardware, persistent collectives could be implemented by directly programming the network hardware with triggered operations, thus reducing the software overhead to a single instruction that begins the preprogrammed sequence and some form of polling or interrupt/event for completion.

5.3 Extension for MPI Endpoints

The MPI Forum is currently considering a proposal to introduce MPI endpoints to the MPI Standard [18]. The proposed API for persistent collective communication naturally extends to an environment that contains multiple MPI endpoints (as defined in the current version of the proposal). However, the anticipated common use-cases for MPI endpoints involve multiple communicating threads within each MPI process - in particular multiple threads concurrently active in collective communication operations using different local endpoints representing different ranks in the same communicator. This usage poses additional challenges to the implementation of both LibNBC and LibPNBC.

For LibNBC, either the schedule cache must be protected for multi-thread concurrent accesses or duplicated for each thread or for each MPI endpoint (depending on the thread support level) to isolate each cache and guarantee only single-thread access. For LibPNBC, each MPI endpoint will create its own `MPI_REQUEST` that will contain the schedule applicable for that endpoint. This is potentially wasteful; endpoints might have to share certain common data.

For implementations of LibNBC and LibPNBC layered on point-to-point, care must be taken to ensure that these messages are matched correctly. Using the same communicator (or a duplicate) with the collective operation should ensure that point-to-point messages can be safely sent to, and received from any MPI endpoint by using the correct rank. The tag would, as in the current implementation, identify which collective operation the message is part of. However, using the parent communicator or `MPI_COMM_WORLD` would be problematic since it would have inadequate addressability to disambiguate all the pair-wise messages.

5.4 Other Future Work

Autotuners (*e.g.,* [14]) could be used to explore the parameter space to choose collective operations on one run, followed by profile-guided optimization for future runs. This substitutes a complex and long-running `_INIT` with a quick table look up for those operations optimized on the initial run. The availability of persistent collectives suggests further optimizations and functionality in the datatypes themselves. Two particular options include the use of "highly optimized data type descriptions" when the operations that use them are persistent (aka data-type crushing; (*cf,* early work on persistent send/receive plus optimized datatypes in [4]). That is, the gather/scatter behavior of complex data types can be studied at greater fixed cost when the reuse of such improved performance can be amortized over many reuses. This is to first order an orthogonal optimization from the choice of the collective algorithm itself. Additionally, so-called optimized or "active datatypes" [2, 8, 15, 20] offer the potential to work with concepts such as circular buffers and other state-machine generated changes in persistent operations, thereby enhancing their generality while retaining enhanced performance over late-binding alternatives.

Chapter 6

Conclusion

This research has presented a design of the first prototype implementation of persistent, collective communication operations in MPI-4. This model implementation, LibPNBC, is based on LibNBC. This initial implementation has been based on Open MPI; porting to other MPI middleware systems remains as future work. Functionality and acceptable performance for a model implementation has been demonstrated. Optimized versions should be able to achieve higher performance than either blocking or nonblocking operations for cases where repetitive use of the same parameters across a group enables the selection of this operation in lieu of the late-binding options currently standardized in MPI-3, and earlier. Additionally, the model implementation serves as a functional basis for supporting the standardization process that is presently being pursued in the MPI Forum.

Chapter 7

Appendix A: APIs

7.1 Persistent Barrier Synchronization

MPI_IBARRIER_INIT(comm, info, request)

IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Ibarrier_init(MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

```
MPI_Ibarrier_init(comm, info, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_IBARRIER_INIT(COMM, INFO, REQUEST, IERROR)
    INTEGER COMM, INFO, REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the barrier operation.

7.2 Persistent Broadcast

MPI_IBCAST_INIT(buffer, count, datatype, root, comm, info, request)

INOUT	buffer	starting address of buffer (choice)
IN	count	number of entries in buffer (non-negative integer)
IN	datatype	data type of buffer (handle)
IN	root	rank of broadcast root (integer)
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Ibcast_init(void* buffer, int count, MPI_Datatype datatype,
             int root, MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

```
MPI_Ibcast_init(buffer, count, datatype, root, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: buffer
    INTEGER, INTENT(IN) :: count, root
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_IBCAST_INIT(BUFFER, COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR)
    <type> BUFFER(*)
    INTEGER COUNT, DATATYPE, ROOT, COMM, INFO, REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the broadcast operation.

7.3 Persistent Gather

MPI_IGATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcount	number of elements for any single receive (non-negative integer, significant only at root)
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Igather_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

```
MPI_Igather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
```

```
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_IGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE,
             ROOT, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
    REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the gather operation.

MPI_IGATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
IN	displs	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)
IN	recvtype	data type of recv buffer elements (significant only at root) (handle)
IN	root	rank of receiving process (integer)
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Igatherv_init(const void* sendbuf, int sendcount,
              MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
              const int displs[], MPI_Datatype recvtype, int root,
              MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

```
MPI_Igatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs,
              recvtype, root, comm, info, request, ierror)
```

```
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, root
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_IGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS, DISPLS,
             RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, ROOT,
    COMM, INFO, REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the gatherv operation.

7.4 Persistent Scatter

MPI_ISCATTER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm, info, request)

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcount	number of elements sent to each process (non-negative integer, significant only at root)
IN	sendtype	data type of send buffer elements (significant only at root) (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Iscatter_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

```
MPI_Iscatter_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, root, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
```

```
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_ISCATTER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
             RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, INFO,
    REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the scatter operation.

MPI_ISCATTERV_INIT(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm, info, request)

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements in receive buffer (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	root	rank of sending process (integer)
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Iscatterv_init(const void* sendbuf, const int sendcounts[],
             const int displs[], MPI_Datatype sendtype, void* recvbuf,
             int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm,
             MPI_Info info, MPI_Request *request)
```

```
MPI_Iscatterv_init(sendbuf, sendcounts, displs, sendtype, recvbuf,
             recvcount, recvtype, root, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
```

```
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), displs(*)
    INTEGER, INTENT(IN) :: recvcount, root
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_ISCATTERV_INIT(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE, RECVBUF,
            RECVCOUNT, RECVTYPE, ROOT, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
    COMM, INFO,REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the scatterv operation.

7.5 Persistent Gather-to-all

MPI_IALLGATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Iallgather_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

```
MPI_Iallgather_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
```

```
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_IALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
            RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
    IERROR
```

Creates a nonblocking, persistent collective communication request for the allgather operation.

MPI_IALLGATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements in send buffer (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length group size) containing the number of elements that are received from each process
IN	displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Iallgatherv_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
             MPI_Info info, MPI_Request* request)
```

```
MPI_Iallgatherv_init(sendbuf, sendcount, sendtype, recvbuf, recvcounts,
             displs, recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
```

```
    INTEGER, INTENT(IN) :: sendcount
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_IALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNTS,
            DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    INFO, REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the allgatherv operation.

7.6 Persistent All-to-All Scatter/Gather

MPI_IALLTOALL_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each process (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcount	number of elements received from any process (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Ialltoall_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

```
MPI_Alltoall_init(sendbuf, sendcount, sendtype, recvbuf, recvcount,
             recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
```

```
TYPE(MPI_Comm), INTENT(IN) :: comm
TYPE(MPI_Info), INTENT(IN) :: info
TYPE(MPI_Request), INTENT(OUT) :: request
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_IALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
            RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
    IERROR
```

Creates a nonblocking, persistent collective communication request for the alltoall operation.

MPI_IALLTOALLV_INIT(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array (of length group size) specifying the number of elements to send to each rank
IN	sdispls	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank
IN	rdispls	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Ialltoallv_init(const void* sendbuf, const int sendcounts[],
             const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
             const int recvcounts[], const int rdispls[],
             MPI_Datatype recvtype, MPI_Comm comm, MPI_info info,
             MPI_Request *request)
```

```
MPI_Ialltoallv_init(sendbuf, sendcounts, sdispls, sendtype, recvbuf,
            recvcounts, rdispls, recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  sendcounts(*), sdispls(*),
    recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE, RECVBUF,
            RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
    RECVTYPE, COMM, INFO, REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the alltoallv operation.

MPI_ALLTOALLW_INIT(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each rank (array of non-negative integers)
IN	sdispls	integer array (of length group size). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for process j (array of integers)
IN	sendtypes	array of datatypes (of length group size). Entry j specifies the type of data to send to process j (array of handles)
OUT	recvbuf	address of receive buffer (choice)
IN	recvcounts	integer array (of length group size) specifying the number of elements that can be received from each rank (array of non-negative integers)
IN	rdispls	integer array (of length group size). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from process i (array of integers)
IN	recvtypes	array of datatypes (of length group size). Entry i specifies the type of data received from process i (array of handles)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Alltoallw_init(const void* sendbuf, const int sendcounts[],
              const int sdispls[], const MPI_Datatype sendtypes[],
```

```
            void* recvbuf, const int recvcounts[], const int rdispls[],
            const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
            MPI_Request *request)

MPI_Alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes, recvbuf,
            recvcounts, rdispls, recvtypes, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), sdispls(*),
    recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
    recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES, RECVBUF,
            RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPES(*), RECVCOUNTS(*),
    RDISPLS(*), RECVTYPES(*), COMM, INFO, REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the alltoallw operation.

7.7 Persistent Reduce

MPI_REDUCE_INIT(sendbuf, recvbuf, count, datatype, op, root, comm, info, request)

IN	sendbuf	address of send buffer (choice)
OUT	recvbuf	address of receive buffer (choice, significant only at root)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	reduce operation (handle)
IN	root	rank of root process (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Reduce_init(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm,
             MPI_Info info, MPI_Request *request)
```

```
MPI_Reduce_init(sendbuf, recvbuf, count, datatype, op, root, comm, info,
             request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  count, root
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_REDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, INFO,
            REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, ROOT, COMM, INFO, REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the reduce operation.

7.8 Persistent All-Reduce

MPI_ALLREDUCE_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	data type of elements of send buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Allreduce_init(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
             MPI_Request *request)
```

```
MPI_Allreduce_init(sendbuf, recvbuf, count, datatype, op, comm, info,
             request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_ALLREDUCE_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO,
             REQUEST, IERROR)
```

```
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the allreduce operation.

7.9 Persistent Reduce-Scatter with Equal Blocks

MPI_REDUCE_SCATTER_BLOCK_INIT(sendbuf, recvbuf, recvcount, datatype, op, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcount	element count per block (non-negative integer)
IN	datatype	data type of elements of send and receive buffers (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Reduce_scatter_block_init(const void* sendbuf, void* recvbuf,
            int recvcount, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm,
            MPI_Info info, MPI_Request *request)
```

```
MPI_Reduce_scatter_block_init(sendbuf, recvbuf, recvcount, datatype, op,
            comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: datatype
    TYPE(MPI_Op), INTENT(IN) :: op
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
MPI_REDUCE_SCATTER_BLOCK_INIT(SENDBUF, RECVBUF, RECVCOUNT, DATATYPE, OP,
            COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the reduce-scatter with equal blocks operation.

7.10 Persistent Reduce-Scatter

MPI_REDUCE_SCATTER_INIT(sendbuf, recvbuf, recvcounts, datatype, op, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array specifying the number of elements in result distributed to each process. Array must be identical on all calling processes.
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Reduce_scatter_init(const void* sendbuf, void* recvbuf,
             const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
             MPI_Comm comm, MPI_Info info, MPI_Request *request)
```

```
MPI_Reduce_scatter_init(sendbuf, recvbuf, recvcounts, datatype, op, comm,
             info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  recvcounts(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
```

```
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_REDUCE_SCATTER_INIT(SENDBUF, RECVBUF, RECVCOUNTS, DATATYPE, OP, COMM,
              INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, INFO, REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the reduce-scatter operation.

7.11 Persistent Inclusive Scan

MPI_SCAN_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in input buffer (non-negative integer)
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	communicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Scan_init(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

```
MPI_Scan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_SCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
            IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the inclusive scan operation.

7.12 Persistent Exclusive Scan

MPI_EXSCAN_INIT(sendbuf, recvbuf, count, datatype, op, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
OUT	recvbuf	starting address of receive buffer (choice)
IN	count	number of elements in input buffer (non-negative integer)
IN	datatype	data type of elements of input buffer (handle)
IN	op	operation (handle)
IN	comm	intracommunicator (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Exscan_init(const void* sendbuf, void* recvbuf, int count,
             MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

```
MPI_Exscan_init(sendbuf, recvbuf, count, datatype, op, comm, info, request,
             ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Datatype), INTENT(IN) ::  datatype
    TYPE(MPI_Op), INTENT(IN) ::  op
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_EXSCAN_INIT(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, INFO, REQUEST,
            IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER COUNT, DATATYPE, OP, COMM, INFO, REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the exclusive scan operation.

7.13 Persistent Neighborhood Gather

MPI_INEIGHBOR_ALLGATHER_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each neighbor (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcount	number of elements received from each neighbor (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Ineighbor_allgather_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, int recvcount,
             MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
             MPI_Request *request)
```

```
MPI_Ineighbor_allgather_init(sendbuf, sendcount, sendtype, recvbuf,
             recvcount, recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN) ::  sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_INEIGHBOR_ALLGATHER_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
             RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
    IERROR
```

Creates a nonblocking, persistent collective communication request for the neighborhood allgather operation.

MPI_INEIGHBOR_ALLGATHERV_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each neighbor (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length indegree) containing the number of elements that are received from each neighbor
IN	displs	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Ineighbor_allgatherv_init(const void* sendbuf, int sendcount,
             MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
             const int displs[], MPI_Datatype recvtype, MPI_Comm comm,
             MPI_Info info, MPI_Request *request)
```

```
MPI_Ineighbor_allgatherv_init(sendbuf, sendcount, sendtype, recvbuf,
             recvcounts, displs, recvtype, comm, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
```

```
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount
    INTEGER, INTENT(IN), ASYNCHRONOUS :: recvcounts(*), displs(*)
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_NEIGHBOR_ALLGATHERV_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
            RECVCOUNTS, DISPLS, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*), RECVTYPE, COMM,
    INFO, REQUEST, IERROR
```

Creates a persistent collective communication request for the neighborhood allgatherv operation.

7.14 Persistent Neighborhood Alltoall

MPI_INEIGHBOR_ALLTOALL_INIT(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcount	number of elements sent to each neighbor (non-negative integer)
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcount	number of elements received from each neighbor (non-negative integer)
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Ineighbor_alltoall_init(const void* sendbuf, int sendcount,
            MPI_Datatype sendtype, void* recvbuf, int recvcount,
            MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info,
            MPI_Request *request)
```

```
MPI_Ineighbor_alltoall_init(sendbuf, sendcount, sendtype, recvbuf,
            recvcount, recvtype, comm, info, request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN) :: sendcount, recvcount
    TYPE(MPI_Datatype), INTENT(IN) :: sendtype, recvtype
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Request), INTENT(OUT) :: request
    TYPE(MPI_Info), INTENT(IN) :: info
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_INEIGHBOR_ALLTOALL_INIT(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
             RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, INFO, REQUEST,
    IERROR
```

Creates a nonblocking, persistent collective communication request for the neighborhood alltoall operation.

MPI_INEIGHBOR_ALLTOALLV_INIT(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement (relative to sendbuf) from which send the outgoing data to neighbor j
IN	sendtype	data type of send buffer elements (handle)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor
IN	rdispls	integer array (of length indegree). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from neighbor i
IN	recvtype	data type of receive buffer elements (handle)
IN	comm	communicator with topology structure (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Ineighbor_alltoallv_init(const void* sendbuf,
            const int sendcounts[], const int sdispls[],
            MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
            const int rdispls[], MPI_Datatype recvtype, MPI_Comm comm,
            MPI_Info info, MPI_Request *request)
```

```
MPI_Ineighbor_alltoallv_init(sendbuf, sendcounts, sdispls, sendtype,
            recvbuf, recvcounts, rdispls, recvtype, comm, info, request,
            ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS ::  recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS ::  sendcounts(*), sdispls(*),
    recvcounts(*), rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN) ::  sendtype, recvtype
    TYPE(MPI_Comm), INTENT(IN) ::  comm
    TYPE(MPI_Info), INTENT(IN) ::  info
    TYPE(MPI_Request), INTENT(OUT) ::  request
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_INEIGHBOR_ALLTOALLV_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
            RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, INFO, REQUEST,
            IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*), RDISPLS(*),
    RECVTYPE, COMM, INFO, REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the neighborhood alltoallv operation.

MPI_INEIGHBOR_ALLTOALLW_INIT(sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm, info, request)

IN	sendbuf	starting address of send buffer (choice)
IN	sendcounts	non-negative integer array (of length outdegree) specifying the number of elements to send to each neighbor
IN	sdispls	integer array (of length outdegree). Entry j specifies the displacement in bytes (relative to sendbuf) from which to take the outgoing data destined for neighbor j (array of integers)
IN	sendtypes	array of datatypes (of length outdegree). Entry j specifies the type of data to send to neighbor j (array of handles)
OUT	recvbuf	starting address of receive buffer (choice)
IN	recvcounts	non-negative integer array (of length indegree) specifying the number of elements that are received from each neighbor
IN	rdispls	integer array (of length indegree). Entry i specifies the displacement in bytes (relative to recvbuf) at which to place the incoming data from neighbor i (array of integers)
IN	recvtypes	array of datatypes (of length indegree). Entry i specifies the type of data received from neighbor i (array of handles)
IN	comm	communicator with topology structure (handle)
IN	info	info argument (handle)
OUT	request	communication request (handle)

```
int MPI_Ineighbor_alltoallw_init(const void* sendbuf,
             const int sendcounts[], const MPI_Aint sdispls[],
```

```
            const MPI_Datatype sendtypes[], void* recvbuf,
            const int recvcounts[], const MPI_Aint rdispls[],
            const MPI_Datatype recvtypes[], MPI_Comm comm, MPI_Info info,
            MPI_Request *request)

MPI_Ineighbor_alltoallw_init(sendbuf, sendcounts, sdispls, sendtypes,
            recvbuf, recvcounts, rdispls, recvtypes, comm, info, request,
            ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS :: sendbuf
    TYPE(*), DIMENSION(..), ASYNCHRONOUS :: recvbuf
    INTEGER, INTENT(IN), ASYNCHRONOUS :: sendcounts(*), recvcounts(*)
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN), ASYNCHRONOUS :: sdispls(*),
    rdispls(*)
    TYPE(MPI_Datatype), INTENT(IN), ASYNCHRONOUS :: sendtypes(*),
    recvtypes(*)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror

MPI_INEIGHBOR_ALLTOALLW_INIT(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPES,
            RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPES, COMM, INFO, REQUEST,
            IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) SDISPLS(*), RDISPLS(*)
    INTEGER SENDCOUNTS(*), SENDTYPES(*), RECVCOUNTS(*), RECVTYPES(*), COMM,
    INFO, REQUEST, IERROR
```

Creates a nonblocking, persistent collective communication request for the neighborhood alltoallw operation.

Bibliography

[1] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Trans. Parallel Distrib. Syst.*, 8(11):1143–1156, 1997.

[2] Alexandra Carpen-Amarie, Sascha Hunold, and Jesper Larsson Träff. On the expected and observed communication performance with MPI derived datatypes. In *Proceedings of the 23rd European MPI Users' Group Meeting, EuroMPI 2016, Edinburgh, United Kingdom, September 25-28, 2016*, pages 108–120, 2016.

[3] Isaías A. Comprés. On-line application-specific tuning with the Periscope tuning framework and the MPI tools interface, 2014. Petascale Tools Workshop.

[4] Rossen Petkov Dimitrov. *Overlapping of communication and computation and early binding: Fundamental mechanisms for improving parallel performance on clusters of workstations.* Mississippi State University, 2001. Ph.D. dissertation, Dept. of Computer Science.

[5] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[6] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 09 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[7] Esthela Gallardo, Jérôme Vienne, Leonardo Fialho, Patricia J. Teller, and James C. Browne. MPI advisor: a minimal overhead tool for MPI library performance tuning. In Jack J. Dongarra, Alexandre Denis, Brice Goglin, Emmanuel Jeannot, and Guillaume Mercier, editors, *Proceedings of the 22nd European MPI Users' Group Meeting, EuroMPI 2015, Bordeaux, France, September 21-23, 2015*, pages 6:1–6:10. ACM, 2015.

[8] Robert Ganian, Martin Kalany, Stefan Szeider, and Jesper Larsson Träff. Polynomial-time construction of optimal MPI derived datatype trees. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 638–647, 2016.

[9] Nathan Hjelm. An evaluation of the one-sided performance in open mpi. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, pages 184–187, New York, NY, USA, 2016. ACM.

[10] T. Hoefler and A. Lumsdaine. Design, Implementation, and Usage of LibNBC. Technical report, Open Systems Lab, Indiana University, Aug. 2006.

[11] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Accurately measuring overhead, communication time and progression of blocking and nonblocking collective operations at massive scale. *International Journal of Parallel, Emergent and Distributed Systems*, 25(4):241–258, 2010.

[12] Sascha Hunold, Alexandra Carpen-Amarie, Felix Donatus Lübbe, and Jesper Larsson Träff. Automatic verification of self-consistent MPI performance guidelines. In *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, pages 433–446, 2016.

[13] MPI Forum. MPI: A message-passing interface standard. Technical report, MPI Forum, Knoxville, TN, USA, 2015.

[14] Anna Sikora, Eduardo César, Isaías Comprés, and Michael Gerndt. Autotuning of mpi applications using ptf. In *Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications*, SEM4HPC '16, pages 31–38, New York, NY, USA, 2016. ACM.

[15] Anthony Skjellum, Daniel J. Holmes, and Purushotham V. Bangalore. Active Datatypes in MPI: Maximizing Performance and Generalizing Persistent Operations for MPI-4, May 2017. Unpublished.

[16] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[17] Jeff Squyres. OpenMPI developer documents. *github.com*, 2016.

[18] Srinivas Sridharan, James Dinan, and Dhiraj D. Kalamkar. Enabling efficient multithreaded MPI communication through a library-based implementation of MPI endpoints. In Trish Damkroger and Jack Dongarra, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pages 487–498. IEEE Computer Society, 2014.

[19] Andrew Lumsdaine Torsten Hoefler and Wolfgang Rehm. Implementation and performance analysis of non-blocking collective operations for MPI. *Supercomputing*, 2007.

[20] Jesper Larsson Träff. A library for advanced datatype programming. In *Proceedings of the 23rd European MPI Users' Group Meeting, EuroMPI 2016, Edinburgh, United Kingdom, September 25-28, 2016*, pages 98–107, 2016.

[21] Jesper Larsson Träff. Practical, linear-time, fully distributed algorithms for irregular gather and scatter. *CoRR*, abs/1702.05967, 2017.

[22] Jesper Larsson Träff, Alexandra Carpen-Amarie, Sascha Hunold, and Antoine Rougier. Message-combining algorithms for isomorphic, sparse collective communication. *CoRR*, abs/1606.07676, 2016.

[23] Jesper Larsson Träff, Antoine Rougier, and Sascha Hunold. Implementing a classic: zero-copy all-to-all communication with mpi datatypes. In Arndt Bode, Michael Gerndt, Per Stenström, Lawrence Rauchwerger, Barton P. Miller, and Martin Schulz, editors, *2014 International Conference on Supercomputing, ICS'14, Muenchen, Germany, June 10-13, 2014*, pages 135–144. ACM, 2014.