Exposition, Clarification, and Expansion of MPI Semantic Terms and Conventions

Is a nonblocking MPI function permitted to block?

Purushotham V. Bangalore
Department of Computer Science
University of Alabama at Birmingham
Birmingham, AL, USA
puri@uab.edu

Rolf Rabenseifner
High-Performance Computing Center
Stuttgart, University of Stuttgart
Stuttgart, Germany
rabenseifner@hlrs.de

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

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

Guillaume Mercier
Bordeaux Institute of Technology
Inria, LaBRI, F-33400 Talence, France
guillaume.mercier@bordeaux-inp.fr

Claudia Blaas-Schenner
VSC Research Center, TU Wien
A-1040 Vienna, Austria
claudia.blaas-schenner@tuwien.ac.at

Anthony Skjellum
Univ. of Tennessee at Chattanooga
Chattanooga, TN, USA
tony-skjellum@utc.edu

ABSTRACT

This paper offers a timely study and proposed clarifications, revisions, and enhancements to the Message Passing Interface's (MPI's) Semantic Terms and Conventions. To enhance MPI, a clearer understanding of the meaning of the key terminology has proven essential, and, surprisingly, important concepts remain underspecified, ambiguous and, in some cases, inconsistent and/or conflicting despite 26 years of standardization. This work addresses these concerns comprehensively and usefully informs MPI developers, implementors, those teaching and learning MPI, and power users alike about key aspects of existing conventions, syntax, and semantics. This paper will also be a useful driver for great clarity in current and future standardization and implementation efforts for MPI.

CCS CONCEPTS

• **Computing methodologies → Parallel computing methodologies**.

KEYWORDS

MPI, message-passing, semantic terms, naming conventions

ACM Reference Format:
Purushotham V. Bangalore, Rolf Rabenseifner, Daniel J. Holmes, Julien Jaeger, Guillaume Mercier, Claudia Blaas-Schenner, and Anthony Skjellum. 2019. Exposition, Clarification, and Expansion of MPI Semantic Terms and Conventions: Is a nonblocking MPI function permitted to block?. In *26th European MPI Users' Group Meeting (EuroMPI 2019), September 11–13, 2019, Zürich, Switzerland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3343211.3343213

EuroMPI 2019, September 11–13, 2019, Zürich, Switzerland

ACM ISBN 978-1-4503-7175-9/19/09…$15.00
https://doi.org/10.1145/3343211.3343213

1 INTRODUCTION

This paper offers a timely study and proposed clarifications, revisions, and enhancements to the Message Passing Interface's (MPI's) Semantic Terms and Conventions. To enhance MPI moving forward, a clearer understanding of the meaning of the key terminology has proven essential. Surprisingly, important concepts remain insufficiently specified, ambiguous, and in some cases, inconsistent or conflicting despite 26 years of standardization, implementation, and utilization. The current draft of MPI-4 [7] (as currently approved by the MPI Forum) is used for the basis of this study.

This paper addresses these concerns comprehensively and will usefully inform MPI developers, implementors, those teaching and learning MPI, and power users alike about key aspects of existing conventions, syntax, and semantics. This paper will also serve as a useful driver for great clarity in current and future standardization and implementation efforts for the Message Passing Interface.

None of the changes proposed is likely to impact major MPI implementations' functionally. Nor do we expect typical applications to have widely misinterpreted the use of MPI. Rather, this paper provides the *de facto* concepts of the MPI Standard as well as a path for these to become the *de jure* syntax, semantics, and conventions in future editions of the MPI Standard. Stipulating that the authors of this paper do not expect to cause practical disruptions with this set of clarifications, these clarifications nonetheless are crucial to avoiding confusion and ambiguity in key aspects of MPI moving forward. Such concerns are likely to grow as we add and improve certain aspects of MPI, and when we seek to bridge gaps to standards interoperability, to new user communities and applications, and into new operating environments such as Exascale systems.

The remainder of this paper is organized as follows: Section 2 describes various semantic terms defined in the different versions of the MPI Standard while identifying certain discrepancies between these terms. Section 3 defines new semantic terms and clarifies the original intent of other currently used semantic terms. Section 4 overviews the semantics of all communication procedures in MPI. Section 5 mentions related work and Section 6 summarizes the paper and identifies future work.

2 EXISTING TERM DEFINITIONS: EXPOSING THE ORIGINAL MEANING

For MPI-3.1 and below, a blocking MPI procedure might not actually block in the traditional sense.

The term **blocking** was first defined in MPI-1.0 as follows: "If return from the procedure indicates the user is allowed to re-use resources specified in the call" [3]. This was later modified in MPI-2.0 to: "A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call" [4]. In common usage, a procedure *blocks* when its return is delayed until some anticipated event takes place or some desired system state is reached. For example, we might expect that a blocking send procedure will return only once the send operation is complete. However, these MPI definitions of the term blocking only discuss usage of parameters given to procedures, they do not include any mention of delaying the return of the procedure until a matching MPI procedure is called at another MPI process. A blocking procedure is permitted to return as soon as the supplied parameters can be reused. Therefore, despite possible expectations derived from commonplace usage of the word 'blocking' in English, a blocking procedure is permitted to: **execute quickly, return before the associated MPI operation is complete, and return before other MPI processes have started the MPI operation**. For example, MPI_BSEND is a blocking procedure that is local. The new definition of blocking is discussed in Sections 3.1.1 and 3.2.9.

For MPI-3.1 and below, a nonblocking MPI procedure might actually block in the traditional sense.

The term **nonblocking** was first defined in MPI-1.0 as follows: "If the procedure may return before the operation completes, and before the user is allowed to re-use resources (such as buffers) specified in the call" [3]. This definition was later modified in MPI-2.1 to read as follows: "A procedure is nonblocking if the procedure may return before the operation completes, and before the user is allowed to reuse resources (such as buffers) specified in the call. A nonblocking request is started by the call that initiates it, e.g., MPI_ISEND. The word complete is used with respect to operations, requests, and communications. An operation completes when the user is allowed to reuse resources, and any output buffers have been updated; i.e. a call to MPI_TEST will return flag = true. A request is completed by a call to wait, which returns, or a test or get status call which returns flag = true. This completing call has two effects: the status is extracted from the request; in the case of test and wait, if the request was nonpersistent, it is freed, and becomes inactive if it was persistent. A communication completes when all participating operations complete" [5].

A new definition was again proposed in MPI-3.1 and now reads as follows: "A procedure is nonblocking if it may return before the associated operation completes, and before the user is allowed to reuse resources (such as buffers) specified in the call. The word complete is used with respect to operations and any associated requests and/or communications. An operation completes when the user is allowed to reuse resources, and any output buffers have been updated" [6].

The term nonblocking is not formally a word in English, but its meaning can be extrapolated as the opposite of *blocking*; that is, it would seem reasonable to assume that nonblocking means will not block or must not block. However, as with the term blocking, these MPI definitions of the term nonblocking only address the usage of parameters given to procedures; they do not include any mention of delaying the return of the procedure until a matching completing MPI procedure is called at the same MPI process. A nonblocking procedure is permitted to return as soon as possible, but also as late as possible. A nonblocking procedure is even permitted to complete its operation before returning. Therefore, despite possible expectations derived from commonplace understanding of the word nonblocking in English, a nonblocking procedure is permitted to: **execute slowly, return once the associated MPI operation is complete, and return only after other MPI processes have started the MPI operation.** The new definition of this term is discussed in Sections 3.1.2 and 3.2.10.

The initialization procedures for persistent collective operations (approved for MPI-4.0) are examples of nonblocking procedures according to the definitions in MPI-1.0–MPI-3.1. However, they are non-local and are therefore classified as blocking procedures in our new nomenclature (see Section 3).

There is no such thing as a persistent MPI procedure; only MPI operations can be persistent.

Thus far, the term **persistent** has not been included in the Terms and Conventions chapter of the MPI Standard. It has only been defined by example as part of the specification of the persistent MPI point-to-point communication functionality. However, the recently accepted functionality for persistent collective communication [7] extends the scope of persistence in MPI and requires a new definition to clarify the usage of the term in the context of MPI. The new definition of this term is discussed in section 3.1.3.

The concept of locality classifying an MPI procedure as **local** or **non-local** has always meant what the English words suggested.

The term **local** was first defined in MPI-1.0 as follows: "If completion of the procedure depends only on the local executing process. Such an operation does not require communication with another user process" [3]. This was later modified in MPI-2.0 to read as follows: "A procedure is local if completion of the procedure depends only on the local executing process" [4]. The MPI definition of the term local follows the English definition: The return of a local procedure depends only on what occurs at the calling MPI process and should not wait for any operations or procedures at any other MPI process.

The term **non-local** was first defined in MPI-1.0 as follows: "If completion of the operation may require the execution of some MPI

procedure on another MPI process. Such an operation may require communication occurring with another user process" [3]. This was later modified in MPI-2.0 as follows: "A procedure is non-local if completion of the operation may require the execution of some MPI procedure on another process. Such an operation may require communication occurring with another user process" [4]. The MPI definition of the term non-local follows the English definition as the opposite of local. The return of a non-local procedure may depend on the execution of another operation or procedure(s) at one or more MPI processes other than itself.

3 NEW TERM DEFINITIONS: CLARIFYING THE ORIGINAL INTENT

The MPI Standard uses the words *function*, *routine*, *procedure*, *procedure call*, and *call* interchangeably to specify the language independent bindings and their corresponding ISO C and Fortran APIs. MPI procedures are defined in terms of MPI operations. However, the MPI Standard fails to define an *operation*. Therefore, we first define an MPI operation, then describe the various stages associated within an operation, and different types of MPI operations and MPI procedures. When describing operations, the MPI Standard also uses the terms *completes*, *started*, *initiates* and *freed* in the definition of other terms (such as nonblocking) without first properly defining them. We offer consistent definitions for these terms as well.

3.1 MPI Operation and its Stages

An **MPI operation** consists of four stages: initialization, starting, completion, and freeing.

Initialization The initialization[1] stage hands over the argument list to the operation but not the content of data buffer(s), if any. The specification of an operation may state that array arguments must not be changed until the operation is freed.

Starting The starting stage hands over the control of the data buffer, if any, to the associated operation.
Note that the term **initiation** in MPI refers to the combination in sequence of the initialization and starting stages.

Completion The completion[1] stage returns control of the content of the data buffer to the application and indicates that output buffers, if any, have been updated.
Note that an MPI operation is **complete** when the MPI procedure implementing the completion stage returns.

Freeing The freeing stage returns control of the rest of the argument list (e.g., the buffer address and array arguments).

3.1.1 Blocking Operation. For a blocking operation, all four stages are combined in a single procedure call (as shown in Figure 1).

3.1.2 Nonblocking Operation. For a nonblocking operation, the initialization and starting stages are combined into a single nonblocking procedure call and the completion and freeing stages are combined into a separate, single procedure call, which can be blocking or nonblocking (as shown in Figure 2).

3.1.3 Persistent Operation. For a persistent operation, all four stages are effected with separate procedure calls, each of which may be blocking or nonblocking (as shown in Figure 3).

3.1.4 Collective Operation. Collective MPI operations (groupwise communications) are available as blocking, nonblocking, or persistent operations.

For nonblocking and persistent collective operations, the completion stage may or may not finish before all processes in the group have started the operation.

Figure 1: Blocking Operations State Transition Diagram

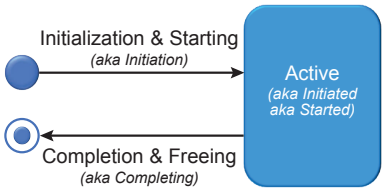

Figure 2: Nonblocking Operations State Transition Diagram[2]

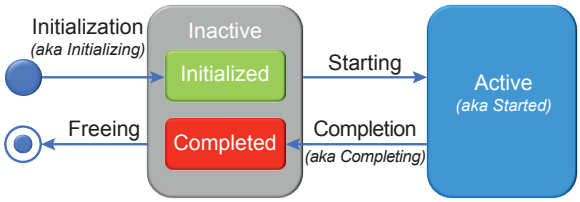

Figure 3: Persistent Operations State Transition Diagram[3]

3.2 MPI Procedure

An MPI operation is implemented as a set of one or more MPI procedures. The semantics of MPI procedures are described using two orthogonal (independent) concepts: completeness (which stages are included) and locality.

MPI procedures can be either incomplete, or completing, or freeing, or completing and freeing based on the status of the associated operation at the time the procedure returns. MPI procedures can either be local or non-local. MPI procedures can also be described as either blocking or nonblocking but these latter two terms refer to combinations of the completeness and locality concepts.

3.2.1 Initialization Procedure. An **MPI procedure is an initialization procedure** if return from the procedure indicates that the associated operation has completed its initialization stage, which implies that the user has handed over control of the argument list (but not the contents of the message buffers) to MPI. The user is still allowed to modify the contents of the data buffers.

[1] We choose *initialization* and *completion* (nouns) intentionally instead of *initializing* and *completing* because the *...ing* forms (gerunds serving as nouns) are only rarely used from Chapters 2 through the end in existing MPI Standard documents. However, where these appear, their meaning is identical to their corresponding parts of speech.

[2] For nonblocking operations we use "completing" in common parlance to refer to the combination of completion and freeing.

[3] We use "completing" in common parlance to refer only to completion for persistent operations; compare Figure 2.

3.2.2 Starting Procedure. An **MPI procedure is a starting procedure** if return from the procedure indicates that the associated operation has completed its starting stage, which implies that the user has handed over control of the data buffers to MPI.

3.2.3 Initiation Procedure. **An MPI procedure is an initiation procedure** if it is both an initialization procedure and a starting procedure, which implies control of the entire argument list is handed over to MPI.

3.2.4 Incomplete Procedure. An **MPI procedure is incomplete** if it may return before the associated operation has finished its completion stage, which implies that the user is not allowed to reuse parameters (such as buffers) specified when initializing the operation. Therefore, an incomplete procedure only includes the initialization and/or[4] starting stages.

3.2.5 Completing Procedure. An **MPI procedure is completing** if return from the procedure indicates that at least one associated operation has finished its completion stage, which implies that the user can rely on the content of the output data buffers and modify the content of input and output data buffers.

If a completing procedure is not also a freeing procedure (see Section 3.2.6) then the user is not permitted to deallocate the data buffers or to modify the array arguments. Procedures not associated with an operation are also defined to be completing.

3.2.6 Freeing Procedure. An **MPI procedure is freeing** if return from the procedure indicates that the associated operation has finished its freeing stage, which implies that the user can reuse all parameters specified when initializing the associated operation.

3.2.7 Local Procedure. An **MPI procedure is local** if it returns control to the calling MPI process based only on the state of the local MPI process that invoked it. Local procedures may be of short or long duration, but their behavior is wholly independent of the activity of other MPI processes or procedure invocations.

3.2.8 Non-local Procedure. An **MPI procedure is non-local** if returning may require the execution of some MPI procedure on another MPI process. Such procedures may require communication occurring with another MPI process.

3.2.9 Blocking Procedure. An **MPI procedure is blocking** if it is completing, and/or freeing, and/or non-local.

3.2.10 Nonblocking Procedure. An **MPI procedure is nonblocking** if it is incomplete and local.

3.2.11 Collective Procedure. An **MPI procedure is collective** if all processes in a process group need to invoke the procedure.

Initialization procedures of collective operations over the same process group must be executed in the same order by all members of the process group.

3.2.12 Synchronizing Procedure. An **MPI collective procedure is synchronizing** if it will only return once all MPI processes (in the associated group of its communicator) have called the same MPI procedure.

[4]We use **and/or** to mean the logical operation inclusive OR throughout this paper.

MPI Operation:	Blocking	Nonblocking	Persistent
1. Initialization	MPI_SEND *(Non-local)* or MPI_SSEND *(Non-local)* or MPI_BSEND *(Local)* or MPI_RSEND *(Local)* or MPI_RECV *(Non-local)* or MPI_BCAST *(Non-local)* *(Completing + Freeing)*	MPI_ISEND *(Local)* or MPI_IRECV *(Local)* or MPI_IBCAST *(Local)* *(Incomplete)*	MPI_SEND_INIT *(Local)* or MPI_RECV_INIT *(Local)* or MPI_BCAST_INIT *(Non-local)* *(Incomplete)*
2. Starting			MPI_START *(Local)* *(Incomplete)*
3. Completion		MPI_WAIT *(Non-local)* *(Completing + Freeing)*	MPI_WAIT *(Non-local)* *(Completing)*
4. Freeing			MPI_REQUEST_FREE *(Local)* *(Freeing)*

Figure 4: The Four Stages of an MPI Operation

The procedures for blocking collective operations and the initialization procedures for persistent collective operations are collective procedures and may or may not be synchronizing. That is, they may or may not return before all processes in the group have called the procedure.

The initiation procedures for nonblocking collective operations and the starting procedures for persistent collective operations are local and shall not be synchronizing.

3.2.13 Common cases and counterexamples. For most communication-related MPI procedures, incomplete procedures are local and completing procedures are non-local. For most nonblocking MPI procedures, an additional prefix letter *I* (an abbreviation of **incomplete** and **immediate**) is included in the procedure name.

Example procedures that show common usage follow:

Nonblocking procedures (incomplete and local):
MPI_ISEND, MPI_IRECV, MPI_IBCAST, MPI_PUT, MPI_GET, MPI-_ACCUMULATE, MPI_IMPROBE, MPI_{SEND|RECV}_INIT, …
Blocking procedures:

- complete and non-local: MPI_SEND, MPI_RECV, MPI_BCAST, MPI_PROBE, …
- incomplete and non-local: MPI_MPROBE, MPI_BCAST_INIT, MPI_FILE_READ_{AT_ALL|ALL|ORDERED}_BEGIN, MPI_FILE_WRITE_{AT_ALL|ALL|ORDERED}_BEGIN,….
- complete and local: MPI_BSEND, MPI_RSEND, MPI_IPROBE, MPI_MRECV.

Figure 4 illustrates how the four different stages of an MPI operation relate to blocking, nonblocking, and persistent operations with some examples of MPI procedures for each type of operation. The MPI_TEST and its variants applied within nonblocking operations are local and incomplete when they produce a false outcome; but are local, completing, and freeing when they produce a true outcome. For persistent operations, MPI_TEST and its variants are local and incomplete when these produce a false outcome; they are local, completing, but not freeing when they produce a true outcome. When multiple requests are involved for MPI_TEST_{ALL|ANY|SOME}, the abovenamed semantics apply request-by-request.

3.2.14 Historical evolution of semantics for procedures. The definition of the MPI semantic term nonblocking, in particular, has been clarified. The *de jure* meaning before MPI-4 was equivalent to the MPI-4 meaning of the MPI semantic term incomplete. But, the *de facto* usage of the term nonblocking, when applied to MPI

procedure calls, has always been the combination of incomplete and local. Thus, the definition of the MPI semantic term blocking has been clarified to be completing, and/or freeing, and/or non-local.

The definition of the MPI semantic term collective has also been clarified. MPI-1 defined blocking collective operations, so all collective procedures were complete and non-local. The *de facto* usage of collective, when applied to MPI procedure calls, was extended in MPI-2 by the inclusion of split collective I/O operations (because the 'begin' procedures are defined to be incomplete and non-local). The *de facto* usage of collective, when applied to MPI procedure calls, was extended in MPI-3 by the inclusion of initiation procedures for nonblocking collective operations, which are defined to be incomplete and local (i.e., they must not be synchronizing).

Figure 5 shows the minimal necessary meaning for collective: technically only a subset of the possible interpretation of the *de jure* meaning, but aligned with the *de facto* and *du jour* (current) meanings, because there were no examples of procedures that required the full extent of the *de jure* meaning. In plain English, the blue box is only in the upper-right quadrant, even though its definition permits it to extend into the lower-right quadrant, because there were no collective, non-local, incomplete procedures in MPI-1.

In Figure 6, the *de facto* and *de jure* meanings coincide. In plain English, MPI-2 added split collective I/O procedures, which were the first examples of collective, non-local, incomplete procedures; that addition requires the blue box to be extended downwards into the lower-right quadrant.

In Figure 7, the *de facto* meaning for collective was further extended such that it actually conflicts with the *de jure* meaning, which was not updated in line with the functionality extensions in MPI-3. In plain English, adding nonblocking collective operations in MPI-3 was done by adding the first examples of procedures defined to be **collective, local, incomplete**. That requires the blue box to be extended leftwards into the lower-left quadrant but with a restriction of "must not be synchronizing." Being local, these procedures must not be synchronizing, which directly conflicts with the strict definition of collective in all versions of the MPI Standard.

In Figure 8, we show that the *de facto* meaning of collective in MPI-4 is unchanged from that of MPI-3. Additional examples of **collective, non-local, incomplete** procedures were added, namely the initialization procedures for persistent collective operations.

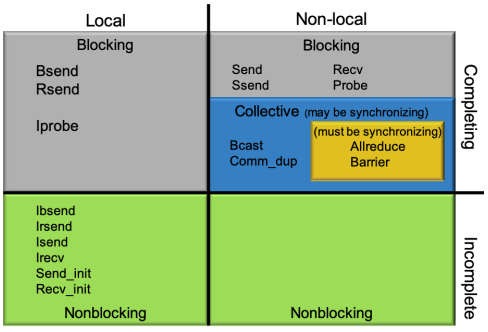

Figure 5: MPI-1 Communication Concepts

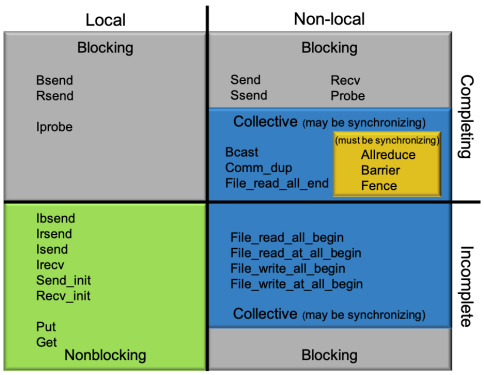

Figure 6: MPI-2 Communication Concepts

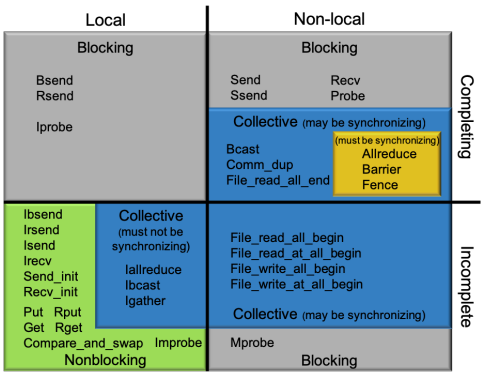

Figure 7: MPI-3 Communication Concepts

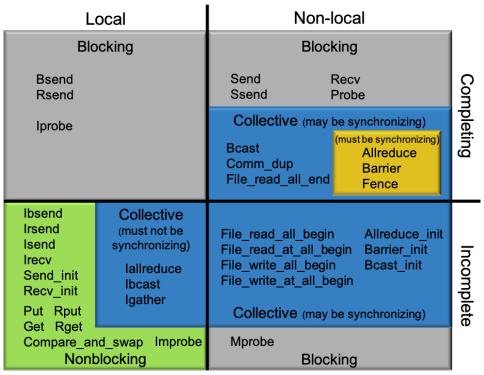

Figure 8: MPI-4 Communication Concepts

4 SEMANTICS OF MPI PROCEDURES

In this section, we break down the semantics of the various communication procedures provided in the MPI Standard chapter-by-chapter. A complete summary of these procedures is provided in Appendix A. We clarify whether these procedures are i) blocking or nonblocking, ii) complete or incomplete, iii) local or non-local, and iv) collective or noncollective. In the case of blocking procedures, we identify the resource(s) that is/are *being blocked*, if any. Figure 8 illustrates the current state of orthogonal MPI communication properties, as discussed earlier.

4.1 Point-to-Point Communication

The Point-to-point chapter includes procedures to support blocking, nonblocking, and persistent communication operations. MPI provides four different send modes: standard, buffered, synchronous, and ready mode [3]. All point-to-point procedures for blocking operations are complete, but some are local and others are non-local. Buffered mode and ready mode send procedures are local – even for blocking operations – because they return without depending on the execution of any MPI procedure in any other MPI process. Hence, MPI_BSEND and MPI_RSEND are two exceptions to the general rule that blocking point-to-point communication routines are both complete and non-local.

All initiation procedures for nonblocking send and receive operations and all initialization procedures for persistent send and receive operations are nonblocking procedures because they are both incomplete and local. In all these cases, the buffers used to send/receive the data are the resources that the user is prohibited to reuse until the corresponding operation is complete. Persistent initialization procedures do not have the *I* prefix in their procedure names even though they are nonblocking procedures and so comprise exceptions to the general rule that the prefix *I* is used to indicate incomplete and immediate for nonblocking procedures. The procedure MPI_BSEND_INIT has an additional *blocked* resource, the attached buffer, in addition to the send buffer, which must not be freed or deallocated before the operation is completed.

The procedure MPI_WAIT, and variants, are non-local. For nonblocking operations, MPI_WAIT performs both the completion and freeing stages of the MPI operation. For persistent operations, MPI_WAIT performs only the completion stage.

The procedure MPI_TEST, and its variants, are local. If MPI_TEST returns flag=FALSE then no stages of the MPI operation were performed. If flag=TRUE is returned, then MPI_TEST will have performed the same operation stages as MPI_WAIT would have done.

The procedures MPI_PROBE, MPI_IPROBE, and MPI_MPROBE are all blocking procedures. MPI_PROBE and MPI_IPROBE are blocking because they are complete by definition since there is no associated MPI operation. MPI_PROBE is non-local while MPI_IPROBE is local. For MPI_IPROBE, the *I* stands for *immediate* and not for *incomplete*. This is one of the exceptions where a complete procedure is local. MPI_MPROBE is blocking because it is non-local; it is also incomplete because the associated receive operation is not complete until a later call to MPI_MRECV. MPI_MPROBE is one of the exceptions where an incomplete procedure is non-local. On the other hand, the MPI_IMPROBE procedure is a nonblocking procedure because it is both incomplete and local.

Two procedures that combine send and receive operations (MPI_SENDRECV and MPI_SENDRECV_REPLACE) are similar to blocking send and receive procedures, which are complete and non-local.

4.2 Collective Communication

The MPI Standard supports three different types of collective operations: blocking, nonblocking, and persistent.

All procedures for blocking collective operations are complete and non-local. In addition, they are collective and may be synchronizing. By way of contrast, all initiation procedures for nonblocking collective operations are incomplete and local. Some nonblocking and persistent operations support variable message length for each MPI process (identified by the initialization procedure having a suffix of *V* or *W*). These operations specify additional resources, such as the array of counts, array of displacements, and/or array of datatypes, which (in addition to the data buffers) cannot be reused until the corresponding operations are complete.

Initialization procedures for persistent collective operations are blocking procedures because they are non-local. Unlike initialization procedures for point-to-point operations, which are nonblocking, local, and not even permitted to communicate, these initialization procedures are collective and may be synchronizing. They are also incomplete; an exception to the general rule that incomplete procedures are local.

The ordering requirement for collective operations applies to the initialization stage only, unless an MPI_INFO assertion is supplied by the user to extend that requirement to include the starting stage as well. Within each communicator, procedures for blocking collective operations, initiation of nonblocking collective operations, and initialization of persistent operations must be called in the same order at all participating processes.

4.3 Communicators

Nearly all procedures described in the "Groups, Contexts, and Communicators" chapter of the MPI Standard that support operations on communicators and groups are blocking procedures that are complete, non-local, and collective. The only exceptions are the two nonblocking constructor operations for communicators; these are initiated by the MPI_COMM_IDUP and MPI_COMM_IDUP_WITH_INFO procedures, which are collective procedures and nonblocking because they are incomplete and local.

4.4 Process Topologies

This chapter provides procedures that create or query virtual topologies and procedures that support blocking, nonblocking, and persistent neighborhood collective communication operations. The procedures for creating a communicator with a virtual topology are blocking because they are complete and non-local. They are also collective and may be synchronizing. Virtual topology query procedures are all blocking because they are complete and local. Also, all procedures for neighborhood collective operations follow the corresponding semantics as described in Section 4.2.

4.5 Process Creation and Management

Process creation and management procedures are blocking and collective; they are complete and non-local, with no exceptions.

4.6 One-Sided Communication

All windows creation procedures (e.g., MPI_WIN_CREATE) are non-local and collective. But, all one-sided communication procedures are nonblocking because they are incomplete and local. However, they lack the *I* prefix in their procedure names. This is another of the exceptions where nonblocking procedures do not include a letter *I* as a prefix in the procedure name. This applies both to one-sided operations that support individual completion by returning an MPI_Request, for example, MPI_RPUT, and to those such as MPI_PUT, where completion is guaranteed by a window synchronization procedure. It also applies to atomic communication operations, such as MPI_COMPARE_AND_SWAP, where the output value is only guaranteed to become valid after completion of the operation during the subsequent window synchronization.

The MPI_FENCE procedure for active-target window synchronization is complete, non-local, collective, and therefore blocking.

The "One-sided Communications" chapter allows the semantics of blocking and nonblocking to vary within compliant implementations. For general active-target synchronization, either MPI_WIN_START, all of the communication functions, or MPI_WIN_COMPLETE, must be non-local, but the standard explicitly allows freedom of implementation to determine which is non-local. For passive-target synchronization, the target process is not involved, so these procedures are local. The MPI_WIN_LOCK and MPI_WIN_LOCK_ALL procedures are incomplete and so nonblocking whereas the MPI_WIN_UNLOCK and MPI_WIN_UNLOCK_ALL procedures are complete and therefore blocking.

Additional study of the semantics and alternative valid implementations of one-sided communication is needed given the freedom offered to implementors by the MPI Standard to choose which operations are non-local and which are not.

4.7 I/O

Data access procedures in the "I/O" chapter of the MPI Standard comprise procedures that support creating, opening, closing, and deleting files, reading data from files, and writing data to files. Data access procedures for reading or writing are categorized according to three orthogonal aspects: positioning, synchronism[5], and coordination. Three types of positioning are supported: explicit offsets (denoted by a suffix of _AT), individual file pointers (with no suffix), shared file pointer (denoted by a suffix of _SHARED or _ORDERED). Three types of synchronism are supported: blocking, nonblocking, and split collective[6]. Two types of coordination are supported: collective (denoted by the suffix _ALL or _ORDERED) and non-collective (no suffix). All I/O chapter procedures specified as non-collective are local because completion is defined only to depend on the local process.

All procedures defined in the I/O chapter as nonblocking have an *I* prefix in their procedure names. However, the *I* prefix appears after MPI_FILE_ and not after MPI_ (as is the case with most nonblocking procedures). The *I* stands for *incomplete*.

Unlike the definitions given in the "Terms and Conventions" chapter of the MPI Standard and used for all other communication procedures, the I/O chapter defines blocking and nonblocking as follows: "A *blocking* I/O call will not return until the I/O request is completed. A *nonblocking* I/O call initiates an I/O operation, but does not wait for it to complete" [6]. The I/O chapter's definition for blocking is a subset of the new definition for blocking given in Section 3. The I/O chapter states that blocking means complete, whereas the new definition states that blocking means completing, *and/or* freeing, *and/or* non-local. Thus, all procedures specified as blocking in the I/O chapter are blocking under both definitions.

But, the I/O chapter's definition for nonblocking is a synonym for incomplete, whereas the new definition given in Section 3 also requires the local semantic. Thus, procedures specified as nonblocking in the I/O chapter are nonblocking if they are local but they are blocking if they are non-local, for example, if they are collective. I/O procedures that are both complete and local include these:

- MPI_FILE_[READ|WRITE]{ |_AT|_SHARED}
- MPI_FILE_{DELETE|SEEK|GET_VIEW}.

I/O procedures that are complete and non-local include these:

- MPI_FILE_[READ|WRITE]_{ALL|AT_ALL|ORDERED}
- MPI_FILE_{OPEN|CLOSE|SYNC}
- MPI_FILE_{SEEK_SHARED|PREALLOCATE}
- MPI_FILE_SET_{VIEW|SIZE|INFO|ATOMICITY}.

The semantics for split collective I/O procedures are described in the MPI Standard (see Section 13.4.5 of MPI-3.1 Standard [6]). The *begin* part of the split collective procedure that starts the corresponding operation and has the suffix _BEGIN in the procedure name is a blocking procedure that is incomplete and non-local. The *end* part of the split collective procedure that completes the operation and has the suffix _END in the procedure name is a blocking procedure that is completing and non-local.

The semantics of collective communication calls defined in Section 4.2 apply to all of collective I/O procedures.

5 RELATED WORK

Several efforts have sought to define formal specifications for the MPI standard [8–11] as well as tools to verify the correctness of MPI programs [12–14]. But, these efforts defined formal specifications for a subset of MPI; moreover, these formal mechanisms are not used by the MPI Forum when adding new functionality nor are these tools used widely in the HPC community. There is broad recognition that formal methods are necessary to define APIs and assist with debugging and verification of HPC applications [8]. While this paper focuses on clarifying and defining the semantic terms used in the MPI standard in plain English without using any formal methods, it nonetheless provides a solid foundation for defining such formal mechanisms and automated tools.

6 CONCLUSION AND FUTURE WORK

This paper provides clarifications, revisions, and enhancements to the current MPI Standard Semantic Terms and Conventions. It is just a first constructive step towards refining and enhancing the semantic terms and conventions found therein. We noted where extant rules apply and where terminology needed to be clarified, such as situations where nonblocking functions apparently are blocking. Several new terms were defined, and the four stages of an MPI operation were enumerated and illustrated. The key takeaway is that we have separated *operation* from *procedure* (aka routine, function, call, function call) and delineated the four stages of an operation. We reviewed the MPI Standard chapter-by-chapter, and described the nature of the procedures defined in each, noting exceptions. Appendix A provides a comprehensive summary of the entirety of all MPI communication procedures semantics, including a table enumerating all classes and cases regarding the four stages of operations comprising a given procedure and remarks concerning the use of resources and blocking status.

Another value of this work is that it could be used, in conjunction with the formal standardization of new MPI operations and procedures, to validate and expose where they fit in MPI's conceptual functionality as illustrated in Figures 5-8. For instance, if there

[5]Ideally, the word 'synchronism', which only appears in the I/O Chapter and only since MPI-2, would be replaced in future with completeness (i.e., 'complete' vs 'incomplete'). This will be apt once split-collectives are removed from the MPI Standard in future.
[6]Split collective syntax is deprecated and will be removed in future. Nonblocking I/O already exists in MPI-3 and is strongly preferred.

were an architecture board for MPI, it could require all proposers to relate their MPI extensions or changes to this existing framework in order to ensure interoperability and clarity before new or modified functionality is accepted for standardization. Barring that, such an analysis could simply become an added, required acceptance procedure under the Forum's current standardization process.

Much work remains in updating the MPI Standard to use these terms consistently across its entirety while identifying other terms that are currently used within it but lack explicit definition and/or require clarification. Further investigation and exposition are also merited to ensure that all MPI terms and conventions are or become consistent and interoperable with other standards such as OpenMP [2], OpenSHMEM [1], and even POSIX [15]. For instance, the meaning of a *process* and *thread* in MPI may differ markedly or subtly from that concept's understanding in POSIX, OpenMP, and OpenSHMEM. We should also consider cross-cutting terminology and parallel constructs of language standards (e.g., C, Fortran, C++).

ACKNOWLEDGMENTS

This work was performed with partial support from the National Science Foundation (NSF) under Grants Nos. CCF-1562306, CCF-1822191, CCF-1821431. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work was part-funded by the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement 801039 (the EPiGRAM-HS project).

The authors acknowledge valuable feedback provided by the members of the MPI Forum and the EuroMPI reviewers.

The authors wish to thank Ms. Holley Beeland for her extensive help with the diagrams presented in this paper.

REFERENCES

[1] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model (PGAS '10)*. ACM, New York, NY, USA, Article 2, 3 pages. https://doi.org/10.1145/2020373.2020375

[2] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* 5, 1 (Jan. 1998), 46–55. https://doi.org/10.1109/99.660313

[3] Message Passing Interface Forum. 1994. *MPI: A Message-Passing Interface Standard. Version 1.0.* Technical Report. Univ. of Tennessee, Knoxville, TN, USA.

[4] Message Passing Interface Forum. 1997. *MPI: A Message-Passing Interface Standard. Version 2.0.* Technical Report. Univ. of Tennessee, Knoxville, TN, USA.

[5] Message Passing Interface Forum. 2008. *MPI: A Message-Passing Interface Standard. Version 2.1.* Technical Report. Univ. of Tennessee, Knoxville, TN, USA.

[6] Message Passing Interface Forum. 2015. *MPI: A Message-Passing Interface Standard. Version 3.1.* Technical Report. Univ. of Tennessee, Knoxville, TN, USA.

[7] Message Passing Interface Forum. 2018. *MPI: A Message-Passing Interface Standard. 2018 Draft Specification.* Technical Report. Univ. of Tennessee, Knoxville, TN, USA. Note: This is the first MPI-4 Draft Specification.

[8] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen F. Siegel, Rajeev Thakur, William Gropp, Ewing L. Lusk, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. 2011. Formal analysis of MPI-based parallel programs. *Commun. ACM* 54, 12 (2011), 82–91. https://doi.org/10.1145/2043174.2043194

[9] Guodong Li, Michael Delisi, Ganesh Gopalakrishnan, and Robert M. Kirby. 2008. Formal Specification of the MPI-2.0 Standard in TLA+. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*. ACM, New York, NY, USA, 283–284.

[10] Guodong Li, Robert Palmer, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby. 2011. Formal specification of MPI 2.0: Case study in specifying a practical concurrent programming API. *Science of Computer Programming* 76, 2 (2011), 65 – 81. https://doi.org/10.1016/j.scico.2010.03.007

[11] Robert Palmer, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby. 2008. An Approach to Formalization and Analysis of Message Passing Libraries. In *Formal Methods for Industrial Critical Systems*, Stefan Leue and Pedro Merino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 164–181.

[12] Stephen F. Siegel. 2007. Model Checking Nonblocking MPI Programs. In *Verification, Model Checking, and Abstract Interpretation*, Byron Cook and Andreas Podelski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–58.

[13] Stephen F. Siegel and George S. Avrunin. 2004. Verification of MPI-Based Software for Scientific Computation. In *Model Checking Software*, Susanne Graf and Laurent Mounier (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 286–303.

[14] Sarvani Vakkalanka, Anh Vo, Ganesh Gopalakrishnan, and Robert M. Kirby. 2009. Reduced Execution Semantics of MPI: From Theory to Practice. In *FM 2009: Formal Methods*, Ana Cavalcanti and Dennis R. Dams (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 724–740.

[15] Stephen R. Walli. 1995. The POSIX Family of Standards. *StandardView* 3, 1 (March 1995), 11–17. https://doi.org/10.1145/210308.210315

A SUMMARY OF SEMANTICS OF ALL MPI COMMUNICATION PROCEDURES

Table Legend:

- **Stages:** i=initialization, s=starting, c=completion, f=freeing
- **Cpl:** ic=incomplete, c=completing, f=freeing procedure
- **Loc:** l=local, nl=non-local
- **Bold**: exceptions, e.g., ic+nl = incomplete+non-local, and c+l = completing+local (both are defined as blocking)
- **Blk:** b=blocking, nb=nonblocking. Note that from a user's view point, this column is only a hint. Relevant is, whether a routine is local or not and which resources are blocked until when. See both previous and last columns.
- **Bold**: exceptions, e.g., nonblocking procedures without prefix "I" or that "I" only marks immediate return.
- **Op:** part of operation type: b-op = blocking operation, nb-op = nonblocking operation, p-op = persistent operation
- **Collective procedures:**
  - C = all processes of the group must call the procedure
  - sq = in the same sequence
  - S1 = blocking synchronization
  - S2 = start-complete-synchronization
- **Blocked resources:** They are blocked after the call until the end of the subsequent stage where this resource is not mentioned further.

Table Footnotes:

1) Musn't return before corresponding MPI receive operation is started.
2) In a correct MPI program, a call to MPI_(I)RSEND requires that the receiver has already started the corresponding receive. Under this assumptions, the call is local.
3) Usually, MPI_WAIT is non-local, but in this case it is local.
4) In case of a MPI_(I)BARRIER, the S1/S2 synchronization is required (instead of "may or may not").
5) Collective: all processes must be completing, but with the free choice of using MPI_WAIT or MPI_TEST returning flag=TRUE.
6) It also may not return until MPI_INIT was called in the children.
7) Addresses are cached on the request handle.
8) One of the rare cases that an incomplete call is non-local and therefore blocking.
9) One shall not free or deallocate the buffer before the operation is freed, that is MPI_REQUEST_FREE returned.
10) For MPI_WAIT and MPI_TEST, see corresponding lines for a) MPI_BSEND, or b) MPI_IBCAST.
11) The prefix "I" marks only that this procedure returns immediately. It is not incomplete.
12) One of the exceptions that a blocking procedure is local.
13) It is complete because it is not associated with an operation.
14) Nonblocking procedure without an "I" prefix.

Procedure	Stages	Cpl	Loc	Blk	Op	Collective			Blocked resources
						C	sq	S1/2	and remarks
MPI_SEND	i-s-c-f	c+f	nl	b	b-op	-			
MPI_SSEND	i-s-c-f	c+f	nl	b	b-op	-			1)
MPI_RSEND	i-s-c-f	c+f	l	b	b-op	-			2) 12)
MPI_BSEND	i-s-c-f	c+f	l	b	b-op	-			12)
MPI_RECV	i-s-c-f	c+f	nl	b	b-op	-			
MPI_ISEND, MPI_ISSEND	i-s----	ic	l	nb	nb-op	-			buffer
MPI_IRECV	i-s----	ic	l	nb	nb-op	-			buffer
corresponding MPI_WAIT	----c-f	c+f	nl		nb-op	-			
corr. MPI_TEST returning flag=TRUE	----c-f	c+f	l		nb-op	-			
corr. MPI_TEST returning flag=FALSE	-------		l		nb-op	-			buffer cached on req
MPI_IBSEND	i-s----	ic	l	nb	nb-op	-			buffer
MPI_IRSEND	i-s----	ic	l	nb	nb-op	-			buffer 2)
corresponding MPI_WAIT	----c-f	c+f	l		nb-op	-			3)
corr. MPI_TEST returning flag=TRUE	----c-f	c+f	l		nb-op	-			
corr. MPI_TEST returning flag=FALSE	-------		l		nb-op	-			buffer 7)
MPI_PROBE	i-s-c-f	c+f	nl	b	b-op	-			13)
MPI_IPROBE	i-s-c-f	c+f	l	b		-			11) 12) 13)
MPI_RECV of a probed message	i-s-c-f	c+f	l	b	b-op	-			12)
MPI_IRECV of a probed message	i-s----	ic	l	nb	nb-op	-			buffer
corresponding MPI_WAIT	----c-f	c+f	l		nb-op	-			3)
corr. MPI_TEST returning flag=TRUE	----c-f	c+f	l		nb-op	-			
corr. MPI_TEST returning flag=FALSE	-------		l		nb-op	-			buffer 7)
MPI_MPROBE	i-s-c-f	ic	nl	b	b-op	-			the message itself 8)
MPI_IMPROBE	i-s-c-f	ic	l	nb	b-op	-			the message itself
MPI_MRECV of a probed message	i-s-c-f	c+f	l	b	b-op	-			12)
MPI_IMRECV of a probed message	i-s----	ic	l	nb	nb-op	-			buffer
corresponding MPI_WAIT	----c-f	c+f	l		nb-op	-			3)
corr. MPI_TEST returning flag=TRUE	----c-f	c+f	l		nb-op	-			
corr. MPI_TEST returning flag=FALSE	-------		l		nb-op	-			buffer 7)
MPI_(-|S|B|R)SEND_INIT, MPI_RECV_INIT	i------	ic	l	nb	p-op	-			buffer address 9) 14)
corresponding MPI_START, MPI_STARTALL	--s----	ic	l	nb	p-op	-			buffer address+content 7),14)
corresponding MPI_WAIT (for (B|R)SEND req.)	----c--	c	l		p-op	-			buffer address 3), 7), 9)
corresponding MPI_WAIT (for other request)	----c--	c	nl		p-op	-			buffer address 7), 9)
corr. MPI_TEST returning flag=TRUE	----c--	c	l		p-op	-			buffer address 7), 9)
corr. MPI_TEST returning flag=FALSE	-------		l		p-op	-			buffer content+address 7)
corr. MPI_REQUEST_FREE (for inactive req-handle)	------f	f	l		p-op	-			
MPI_CANCEL of nonblock./persistent pt-to-pt			l		p-op	-			
MPI_SENDRECV(_REPLACE)	i-s-c-f	c+f	nl	b	b-op	-			

Procedure	Stages	Cpl	Loc	Blk	Op	Collective			Blocked resources
						C	sq	S1/2	and remarks
MPI_BCAST and others	i-s-c-f	c+f	nl	b	b-op	C	sq	S1	4)
MPI_IBCAST and others	i-s----	ic	l	nb	nb-op	C	sq		buffer 4)
MPI_IGATHER**V** and other **…V / …W**	i-s----	ic	l	nb	nb-op	C	sq		buffer, array arguments
corresponding MPI_WAIT	----c-f	c+f	nl		nb-op	C		S2	4) 5)
corr. MPI_TEST returning flag=TRUE	----c-f	c+f	l		nb-op	C		S2	4) 5)
corr. MPI_TEST returning flag=FALSE	-------		l		nb-op				buffer, array arguments 7)
MPI_BCAST_INIT and others	i------	**ic**	**nl**	**b**	p-op	C	sq	S1	buffer address **8)** 9)
MPI_GATHER**V**_INIT and other **…V/…W**_INIT	i------	**ic**	**nl**	**b**	p-op	C	sq	S1	buffer address, array arguments **8)** 9)
corresponding MPI_START, MPI_STARTALL	--s----	ic	l	**nb**	p-op	C			buffer addr.+content, 4,7), **14)**
corresponding MPI_WAIT	----c--	c	nl		p-op	C		S2	buffer address and array arguments cached on the reqest handle, 4,5,7,9)
corr. MPI_TEST returning flag=TRUE	----c--	c	l		p-op	C		S2	buf-addr&arr-args 4,5,7,9)
corr. MPI_TEST returning flag=FALSE	-------		l		p-op				buf addr+content&arr-args 7)
corr. MPI_REQUEST_FREE	------f	f	l		p-op				
MPI_COMM_CREATE	i-s-c--	c	nl	b	b-op	C	sq	S1	coll. over comm arg.
MPI_COMM_CREATE_GROUP	i-s-c--	c	nl	b	b-op	C	sq	S1	coll. over group arg.
MPI_COMM_DUP, MPI_COMM_DUP_WITH_INFO, MPI_COMM_SPLIT, MPI_COMM_SPLIT_TYPE,									
MPI_CART_CREATE, MPI_GRAPH_CREATE, MPI_DIST_GRAPH_CREATE_ADJACENT, MPI_DIST_GRAPH_CREATE,									
MPI_CART_SUB: see MPI_COMM_CREATE									
MPI_INTERCOMM_CREATE, MPI_INTERCOMM_MERGE	i-s-c--	c	nl	b	b-op	C	sq	S1	coll. over union of local & remote group
MPI_COMM_IDUP	i-s----	ic	l	nb	nb-op	C	sq		communicator handle
corresponding MPI_WAIT	----c--	c	nl		nb-op	C		S2	5)
corr. MPI_TEST returning flag=TRUE	----c--	c	l		nb-op	C		S2	5)
corr. MPI_TEST returning flag=FALSE	-------		l		nb-op				
MPI_COMM_FREE	------f	f	nl	b	b-op	C	sq	S1	see, 6.4.3, Adv. to impl.
MPI_INIT, MPI_INIT_THREAD	i-s-c-f	c+f	nl	b	b-op	C	sq	S1	collective over MPI_COMM_WORLD
MPI_FINALIZE	i-s-c-f	c+f	nl	b	b-op	C	sq	S1	collective over all connected processes
MPI_COMM_SPAWN, …._MULTIPLE	i-s-c-f	c+f	nl	b	b-op	C	sq	S1	collective over comm, 6)
MPI_COMM_ACCEPT, MPI_COMM_CONNECT	i-s-c-f	c+f	nl	b	b-op	C	sq	S1	collective over comm
MPI_PUT, MPI_GET, MPI_ACCUMULATE	--s----	ic	l	**nb**	nb-op	-			buffer **14)**
Other one-sided procedures									See corresponding chapter
MPI_FILE_READ/WRITE[_AT|SHARED], MPI_FILE_DELETE/SEEK/GET_VIEW	i-s-c-f	**c+f**	**l**	**b**	b-op	-			**12)**
MPI_FILE_READ/WRITE[_AT]**_[ALL|ORDERED]**, MPI_FILE_OPEN/CLOSE/SEEK_SHARED, MPI_FILE_PREALLOCATE/SYNC, MPI_FILE_SET_VIEW/SIZE/INFO/ATOMICITY	i-s-c-f	c+f	nl	b	b-op	C	sq	S1	
MPI_FILE_**I**READ/**I**WRITE[_AT|SHARED]	i-s----	ic	l	nb	nb-op	-			buffer 10a)
MPI_FILE_**I**READ/**I**WRITE[_AT]**_ALL**	i-s----	ic	l	nb	nb-op	C	sq		buffer 10b)
MPI_FILE_READ/WRITE[_AT]_ALL|ORDERED**_BEGIN**	i-s----	**ic**	**nl**	**b**	b-op	C	sq	S1	buffer **8)**
MPI_FILE_READ/WRITE[_AT]_ALL|ORDERED**_END**	----c-f	c+f	nl	b	b-op	C	sq	S1	