Annex A

Glossary

begin discussion >————————

The following Annex is provided to promote precise usage of MPI concepts and specialized terms, to provide a convenient mechanism to learn MPI nomenclature (especially for new users), and to help ensure consistency in use of functions, operations, etc.

Glossary Inclusion Criteria The criteria for Glossary inclusion are
The term is associated with version MPI-3 ballot xx.
The term is used at least once in the document.
The term is well defined and has a clear context.
Terms that are used in different ways in the MPI-3 spec will have multiple entries.
Restatement of definitions found within the specification are allowed.
Motivation is included within the definition when helpful.
General terms used in accordance with the standard definition are included when deemed important to MPI-3 (e.g., thread safe).

————————< **end discussion**

- **absolute address**
  Displacements relative to MPI_BOTTOM, the start of the address space. Note that MPI_BOTTOM may be viewed as a "zero address" but need not be zero. Refer to Section 2.5.4 on page 14.
- **access epoch**
  The period during which a target window can be accessed by RMA operations. (See also exposure epoch and RMA.) Refer to Section 11.4 on page 369.
- **active**
  The property of a process within a parallel procedure that belongs to a group that may collectively execute the procedure, and some member of that group is currently

executing the procedure code. If a parallel procedure is active in a process, then this process may be receiving messages pertaining to this procedure, even if it does not currently execute the code of this procedure. Refer to Section 6.9 on page 264.

- **active target communication**
  An RMA communication where data is moved from the memory of one process to the memory of another, and both are explicitly involved in the synhcronization. This communication pattern is similar to message passing, except that all the data transfer arguments are provided by one process, and the second process only participates in the synchronization. (See also passive target communication and RMA.).) Refer to Section 11.4 on page 369.

- **associative**
  The property of a collective reduction operation that the order in which the operations are performed does not matter as long as the sequence of the operands is not changed. Binary operation $\otimes$ is associative iff:

$$(A \otimes B) \otimes C = A \otimes (B \otimes C) \tag{A.1}$$

  Reduction operation properties apply to MPI_OP_CREATE argument MPI_Op *op. (See also commutative.) Refer to Section 5.9.5 on page 171.

- **attributes**
  Arbitrary pieces of information cached on three kinds of MPI objects: communicators, windows and datatypes. (See also caching.) Refer to Section 6.7 on page 246.

- **basic datatype**
  A basic datatype is a named predefined datatype, which corresponds to the basic datatypes of the host language. Examples include MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_COMPLEX, MPI_LOGICAL, and MPI_CHARACTER. (See also derived datatype .) Refer to Section 2.4 on page 11.

- **blocking**
  A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call. Refer to Section 2.4 on page 11, and 3.4 on page 38 and 3.2 on page 26.

- **broadcast**
  A collective operation which communicates data from a specified root process to all processes within the group; initially just the first process contains the data, but after the broadcast completes successfully all processes contain it. Refer to Section 5.1 on page 131.

- **buffered communication mode**
  A point-to-point communication protocol in which the send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. However, unlike the standard send, this operation is **local**, and its completion does not depend on the occurrence of a matching receive. Thus, if a send is executed and no matching receive is posted, then MPI must buffer the outgoing message, so as to allow the send call to complete. An error will occur if there is

insufficient buffer space. The amount of available buffer space is controlled by the user. Buffer allocation by the user may be required for the buffered mode to be effective. (See also standard communication mode, synchronous communication mode, ready communication mode.) Refer to Section 3.4 on page 38.

- **caching**
  To store data in a manner for quick re-use later. MPI provides a "caching" facility that allows an application to attach arbitrary pieces of information, called **attributes**, to three kinds of MPI objects: communicators, windows and datatypes. More precisely, the caching facility allows any library to do the following:
  - pass information between calls by associating it with an MPI intra- or inter-communicator, window or datatype,
  - quickly retrieve that information, and
  - be guaranteed that out-of-date information is never retrieved, even if the object is freed and its handle subsequently reused by MPI.

  (See also attributes.) Refer to Section 6.7 on page 246.

- **cartesian topology**
  A virtual ordering of the processes according to a cartesian grid. Process coordinates in a Cartesian structure begin their numbering at 0. Row-major numbering is always used for the processes in a Cartesian structure. This means that, for example, the relation between group rank and coordinates for four processes in a (2×2) grid is as follows.

  coord (0,0): rank 0
  coord (0,1): rank 1
  coord (1,0): rank 2
  coord (1,1): rank 3

  (See also topology and graph topology.) Refer to Section 7 on page 267.

- **client**
  The member of a client/server relationship that makes the request. The *client* connects to the server. MPI provides a mechanism for two sets of MPI processes that do not share a communicator to establish communication. Establishing contact between two groups of processes that do not share an existing communicator is a collective but asymmetric process. (See also server, port name, and service name). Refer to Section 10.4 on page 340.

- **collective call**
  A property of a procedure that all processes in a process group need to invoke the procedure. A collective call may or may not be synchronizing. Collective calls over the same communicator must be executed in the same order by all members of the process group. (See also collective communication.) Refer to Section 2.4 on page 11.

- **collective communication**
  communication that involves a group or groups of processes. Examples include MPI_BARRIER, MPI_BCAST, MPI_GATHER, MPI_GATHERV, MPI_SCATTER, MPI_SCATTERV, MPI_ALLGATHER, MPI_ALLGATHERV, MPI_ALLTOALL,

MPI_ALLTOALLV, MPI_ALLTOALLW, MPI_ALLREDUCE, MPI_REDUCE, MPI_IREDUCE_SCATTER, MPI_SCAN, and MPI_EXSCAN. (See also collective call.) Refer to Chapter 5 on page 131.

- **committed datatype**
  The second step in preparing a *derived datatype* for communication (after "created datatype"). There is no need to commit basic datatypes. They are "pre-committed." (See also created datatype.) Refer to Section 4.1.9 on page 99.

- **communicator**
  A group with the same communication context. (See also communication context.) Refer to Section 3.2.3 on page 29 and 6 on page 209.

- **communication context**
  A prescribed "communication universe" that accommodates messages to be received within the *communication context* they were sent, and messages sent in different *communication contexts* do not interfere. The communicator also specifies the set of processes that share this communication context. Contexts provide the ability to have a separate safe "universe" of message-passing between the two groups. A context is akin to an additional tag that differentiates messages. The system manages this differentiation process. The use of separate communication contexts by distinct libraries (or distinct library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications on "other" communicators, and avoids the need to synchronize entry or exit into library code. Pending point-to-point communications are also guaranteed not to interfere with collective communications within a single communicator. Refer to Section 3.2.3 on page 29, and to Section 6 on page 209.

- **commutative**
  The property of a collective reduction that changing the order of the operands does not change the end result. Binary operation $\odot$ is commutative iff:

$$A \odot B = B \odot A \tag{A.2}$$

  Reduction operation properties apply to MPI_OP_CREATE argument MPI_Op *op. (See also associative.) Refer to Section 5.9.5 on page 171.

- **completion/completed**
  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 MPI_WAIT, which returns, or a MPI_TEST or MPI_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 non persistent, it is freed, and becomes inactive if it was persistent. A communication completes when all participating operations complete. Refer to Section 2.4 on page 11.

- **contiguous**
  A collection of memory locations that are adjacent to one another without intervening data. Refer to Section 4.1.2 on page 79.

- **conversion**
  See **type conversion** and **type conversion**.

- **correct program**
  A program that performs as intended by the program developer; A program that is free of bugs. For example, a **correct program** that utilizes collective communications must invoke collective communications so that deadlock will not occur, whether collective communications are synchronizing or not. (See also erroneous program.) Refer to Section 5.13 on page 200.

- **created datatype**
  The initial step in preparing a *derived datatype* (before "committed datatype"). (See also committed datatype.) Refer to Section 4.1.9 on page 99.

- **data conversion**
  See **type conversion** and **type conversion**.

- **datatype**
  A *datatype* defines a mechanism to describe any data layout, e.g., an array of structures in the memory, which can be used as a message send or receive buffer. (See also basic datatype and derived datatype.) Refer to Chapter 4 on page 77.

- **datatype handle**
  A datatype handle is use to describe the buffer in a communication call. Refer to Section 4.1.9 on page 99.

- **datatype, optional**
  A type that is not required by conforming implementations of MPI-3 (e.g., MPI_INTEGER2). Refer to Section 4.1.9 on page 99.

- **deprecated**
  Constructs that continue to be part of the MPI standard, as documented in Chapter 15, but that users are recommended not to continue using, since better solutions were provided with the current standard. Note that deprecated constructs may eventually be withdrawn. For example, the Fortran binding for MPI-1 functions that have address arguments uses `INTEGER`. This is not consistent with the C binding, and causes problems on machines with 32 bit `INTEGER`s and 64 bit addresses. Refer to Section 2.6.1 on page 16.

- **derived datatype**
  A derived datatype is any datatype that is not predefined. They are opaque objects that specifies two things:

  – A sequence of basic datatypes
  – A sequence of integer (byte) displacements

  The displacements are not required to be positive, distinct, or in increasing order. Therefore, the order of items need not coincide with their order in store, and an item may appear more than once. (See also general datatype, basic datatype and typemap.) Refer to Section 2.4 on page 11 and Chapter 4 on page 77.

- **displacement, file**
  A file *displacement* is an absolute byte position relative to the beginning of a file. The displacement defines the location where a *view* begins. Note that a "file displacement" is distinct from a "typemap displacement." Refer to Section 13.1.1 on page 411.

- **end of file**
  See file size.

- **equivalent datatype**
  Two datatypes are equivalent if they appear to have been created with the same sequence of calls (and arguments) and thus have the same typemap. Two equivalent datatypes do not necessarily have the same cached attributes or the same names. Refer to Section 2.4 on page 11.

- **erroneous program**
  A program that does not perform as intended by the program developer; A program that contains bugs. (See also correct program.) Refer to Section 5.13 on page 200 and Section 2.8.

- **error class**
  A construct designed to allow implementations freedom with their choice of error codes while at the same time provide a predefined subset of error codes to permit an application to classify a specific error code. This is done to allow an implementation to provide as much information as possible in the error code (for use with MPI_ERROR_STRING). To make it possible for an application to interpret an error code, the routine MPI_ERROR_CLASS converts any error code into one of a small set of standard error codes, called *error classes*. Valid error classes are shown in Table 8.1 and Table 8.2. Refer to Section 8.3 on page 298.

- **etype**
  An *etype* (*elementary* datatype) is the unit of data access and positioning for file I/O. It can be any MPI predefined or derived datatype. Derived etypes can be constructed using any of the MPI datatype constructor routines, provided all resulting typemap displacements are non-negative and monotonically nondecreasing. Data access is performed in etype units, reading or writing whole data items of type etype. Offsets are expressed as a count of etypes; file pointers point to the beginning of etypes. Depending on context, the term "etype" is used to describe one of three aspects of an elementary datatype: a particular MPI type, a data item of that type, or the extent of that type. Refer to Section 13.1.1 on page 411.

- **exscan operation**
  A collective procedure in which a partial reduction operation is performed on data supplied by the members of a group. In the exclusive scan (see MPI_EXSCAN), the prefix reduction on process i only includes data up to i-1. (See also scan operation and reduce operation.) Refer to Section 5.9 on page 162.

- **exposure epoch**
  The period during which a target window can be accessed by RMA operations in *active target* communication. (See also access epoch, active target and RMA.) Refer to Section 11.4 on page 369.

- **extent**
  The **extent** of a datatype is defined to be the span from the first byte to the last byte occupied by entries in this datatype, rounded up to satisfy alignment requirements. That is, if

  $$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

  then

  $$\begin{aligned} lb(Typemap) &= \min_j disp_j, \\ ub(Typemap) &= \max_j (disp_j + sizeof(type_j)) + \epsilon, \text{ and} \\ extent(Typemap) &= ub(Typemap) - lb(Typemap). \end{aligned} \tag{A.3}$$

  If $type_i$ requires alignment to a byte address that is a multiple of k_i, then ϵ is the least non-negative increment needed to round $extent(Typemap)$ to the next multiple of $\max_i k_i$. For datatypes that have a "hole" at its beginning or its end, or a datatype with entries that extend above the upper bound or below the lower bound, then

  $$extent(Typemap) = ub(Typemap) - lb(Typemap)$$

  (See also lower bound, upper bound, and true extent.) Refer to Section 4.1 on page 77.

- **external32**
  External32 is a data representation format/specification. The data on the storage medium is always in this canonical External32 representation, and the data in memory is always in the local process's native representation. (See also native and internal.) Refer to Section 13.5 on page 450.

- **fairness**
  The property of parallel and distributed systems that no process is starved, and all processes are accorded the same priority in allowing their accesses to shared resources. When fairness is imposed, all processes have the chance to make progress regardless of what other processes may be doing at the same time. Note that MPI makes no fairness guarantees. Suppose that a send is posted. Then it is possible that the destination process repeatedly posts a receive that matches this send (e.g., using tag MPI_ANY_SOURCE), yet the message is never received, because in each case it is overtaken by another message, sent from another source. Similarly, suppose that a receive was posted by a multi-threaded process. Then it is possible that messages that match this receive are repeatedly received, yet the receive is never satisfied, because it is overtaken by other receives posted at this process (by other executing threads). It is the programmer's responsibility to prevent starvation in such situations. Refer to Section 3.5 on page 42.

- **file**
  An MPI file is an ordered collection of typed data items. MPI supports random or sequential access to any integral set of these items. A file is opened collectively by a group of processes. All collective I/O calls on a file are collective over this group. Refer to Section 13.1.1 on page 411.

- **file consistency**
  Consistency semantics define the outcome of multiple accesses to a single file. All file accesses in MPI are relative to a specific file handle created from a collective open. MPI provides three levels of consistency: sequential consistency among all accesses using a single file handle, sequential consistency among all accesses using file handles created from a single collective open with atomic mode enabled, and user-imposed consistency among accesses other than the above. Sequential consistency means the behavior of a set of operations will be as if the operations were performed in some serial order consistent with program order; each access appears atomic, although the exact ordering of accesses is unspecified. User-imposed consistency may be obtained using program order and calls to MPI_FILE_SYNC. Refer to Section 13.6 on page 459.

- **file handle**
  A reference to an opaque structure that enables a program to access a particular file. A *file handle* is an opaque object created by MPI_FILE_OPEN and freed by MPI_FILE_CLOSE. All operations on an open file reference the file through the file handle. Refer to Section 13.1.1 on page 411.

- **file interoperability**
  file interoperability is the ability to correctly interpret the representation of information previously written to a file. MPI supports the External32 data representation (Section 13.5.2, page 453) as well as the data conversion functions (Section 13.5.3, page 454). Refer to Section 13.5 on page 450.

- **file pointer**
  A *file pointer* is an implicit offset maintained by MPI. "Individual file pointers" are file pointers that are local to each process that opened the file. A "shared file pointer" is a file pointer that is shared by the group of processes that opened the file. Refer to Section 13.1.1 on page 411.

- **file size**
  The *size* of an MPI file is measured in bytes from the beginning of the file. A newly created file has a size of zero bytes. If MPI_FILE_SET_SIZE has resized the file past the current file size, the values of data in the new regions in the file (those locations with displacements between old file size and the specified new file **size**) are undefined. It is implementation dependent whether the MPI_FILE_SET_SIZE routine allocates file space—use MPI_FILE_PREALLOCATE to force file space to be reserved. Refer to Section 13.1.1 on page 411.

- **file view**
  See view, file .

- **filetype**
  A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype. In addition, the extent of any hole in the filetype must be a multiple of the etype's extent. The displacements in the typemap of the filetype are not required to be distinct, Refer to Section 13.1.1 on page 411.

- **gather**
  A collective operation in which one message is sent from n processes in the group to the root. Messages are concatenated in rank order, and the resulting message is received by the root as if by a call to MPI_RECV(recvbuf, recvcount·n, recvtype, ...). Refer to Section 5.1 on page 131 and Section 5.5 on page 139.

- **general datatype**
  see derived datatype.

- **generalized request**
  The request object associated with a user-defined non-blocking operation. Refer to Section 12.1 on page 395 **??** on page 395.

- **global**
  Referring to all members of a group. Refer to Section 5.1 on page 131.

- **graph topology**
  A virtual ordering of the processes according to a generalized graph. (See also topology and cartesian topology.) Refer to Section 7 on page 267.

- **groups**
  Groups define an ordered collection of processes, each with a rank, and it is this group that defines the low-level *identifiers* for inter-process communication (ranks are used for sending and receiving). Thus, groups define a scope for process names in point-to-point communication. Groups may be manipulated separately from communicators in MPI. While it is not permissible to communicate over just a group, a group may be used in one-sided communications to restrict the members associated with an epoch. Each process in the group is assigned a rank between `0` and `n-1`. (See also topology .) Refer to Section 3.2.3 on page 29 and Chapter 6 on page 209.

- **handle**
  A value that enables a program to access a particular opaque structure. Opaque structures are used in MPI for communicators, files, requests, etc. Refer to Section 2.5.1 on page 12.

- **implementation**
  A specific fulfillment of a specification. Refer to Section 2.4 on page 11.

- **IN**
  An argument of an MPI procedure call with the following property: the call may use the input value but does not update the argument. (See also INOUT and OUT.) Refer to Section 2.4 on page 11.

- **INOUT**
  An argument of an MPI procedure call with the following property: the call may both use and update the argument. (See also IN and OUT.) Refer to Section 2.4 on page 11.

- **in place**
  Communication in which the input buffer is reused as the output buffer. This is specified by providing a special argument value, MPI_IN_PLACE, instead of the send

buffer or the receive buffer argument, depending on the operation performed. Refer to Section 5.2 on page **??**.

- **intercommunicator**
  A communicator that identifies two distinct groups (local and remote) of processes linked with a context. (See also intracommunicator.) Refer to Section 5.2 on page 134 and Section 6.6 on page 238.

- **interface**
  Syntax and semantics for invoking services from within an executing application. Refer to Section 2.4 on page 11.

- **internal data representation**
  Data in *internal* data representation can be used for I/O operations in a homogeneous or heterogeneous environment; the implementation will perform type conversions if necessary. The implementation is free to store data in any format of its choice, with the restriction that it will maintain constant extents for all predefined datatypes in any one file. The environment in which the resulting file can be reused is implementation-defined and must be documented by the implementation. (See also native data representation and external32.) Refer to Section 13.5 on page 450.

- **intracommunicator**
  A communicator in which the communication domain does not extend beyond the members of a single local group. (See also intercommunicator.) Refer to Section 5.2 on page 134.

- **local group**
  The recipients of a procedure for an intercommunicator. (See also intercommunicator and remote group.) Refer to Section 5.2 on page 134.

- **local operation**
  A procedure is local if it returns irrespective of the status of other processes. (See also non-local operation.) Refer to Section 2.4 on page 11.

- **lower bound, datatype**
  The displacement of the lowest unit of addressable memory within the datatype. (See also upper bound and extent.) Refer to *lb(Typemap)* in Section 4.9 on page 96.

- **matching**
  (a) A language type (e.g., float) matches an MPI datatype (e.g., MPI_FLOAT). (b) Two datatypes match if their type signatures are identical. (c) A message matches a receive operation, if (c.1) the communicators are identical, (c.2) the source ranks either are identical or a wildcard source was specified by the receive operation, and (c.3) the tags are identical or a wildcard tag was specified by the receive operation. Refer to Section 3.3.1 on page 34.

- **message envelope**
  The non-data portion of a message that can be used to distinguish messages and selectively receive them. This information consists of a fixed number of fields, which are collectively called the **message envelope**. These fields are, source, destination, tag, communicator, and length. Refer to Section 3.2.3 on page 29.

- **native data representation**
  A format for storing data. Data in *native* representation is stored in a file exactly as it is in memory. The advantage of this data representation is that data precision and I/O performance are not lost in type conversions with a purely homogeneous environment. The disadvantage is the loss of transparent interoperability within a heterogeneous MPI environment. (See also internal data representation and external32.) Refer to Section 13.5 on page 450.

- **non-local operation**
  The property of a procedure that completion of the procedure may require the execution of some MPI procedure on another process. Such a procedure may require communication occurring with another user process. That is, the return of the procedure is dependent on other processes. (See also local operation.) Refer to Section 2.4 on page 11.

- **non-overtaking**
  The requirement that if a sender sends two messages in succession to the same destination, and both match the same receive, then the receiver cannot receive the second message if the first one is still pending. Refer to Section 3.5 on page 42.

- **nonblocking**
  The property of a procedure that it 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. (See also completion/completed) Refer to Section 2.4 on page 11 and Section 3.7 on page 48.

- **nondeterminism**
  A nondeterministic program is one in which either (a) repeated executions of the program with the same input may yield different results (weak nondeterminism); or (b) the sequence of states through which the program passes is not uniquely determined by the input even if the results are the same (strong nondeterminism). Nondeterminism may originate from the use of wildcards, MPI_CANCEL, MPI_WAITANY, rounding errors in floating-point reduction operations, and so on. Refer to "message order" in Section 3.5 on page 42.

- **null handle**
  A handle with a reserved value indicating that it refers to no object. (See also null process) Refer to Section 3.7.3 on page 53.

- **null process**
  A "dummy" source or destination (MPI_PROC_NULL) used for communication. This simplifies the code that is needed for dealing with boundaries, for example, in the case of a non-circular shift done with calls to send-receive. The special value MPI_PROC_NULL can be used instead of a rank wherever a source or a destination argument is required in a call. A communication with process MPI_PROC_NULL has no effect. (See also null handle) Refer to Section 3.11 on page 76.

- **null request**
  A request handle with a reserved value (MPI_REQUEST_NULL) indicating that it refers to no object. (See also null handle) Refer to Section 3.7.3 on page 53.

- **offset**
  An *offset* is a position in the file relative to the current view, expressed as a count of etypes. Holes in the view's filetype are skipped when calculating this position. Offset 0 is the location of the first etype visible in the view (after skipping the displacement and any initial holes in the view). For example, an offset of 2 for process 1 in Figure **??** is the position of the 8th etype in the file after the displacement. An "explicit offset" is an offset that is used as a formal parameter in explicit data access routines. Refer to Section 13.1.1 on page 411.

- **one-sided communication**
  See RMA. Refer to Chapter 11 on page 357.

- **opaque**
  Data structures managed by MPI whose size and shape are not visible to the user. Opaque structures are accessed by the application program through **handles**, e.g., a communicator handle. Refer to Section 2.5.1 on page 12.

- **order**
  The arrangement of operations relative to each other according to sequence of invocation. MPI places several requirements upon order: (a) Point-to-point messages are non-overtaking; (b) Nonblocking communication operations are ordered to the execution order; (c) Collective operations must be exectued in the same order at all members of the communication group. Refer to Section 3.7.4 on page 56.

- **origin process**
  The process that performs the call in one-sided communications. (See also RMA and target process) Refer to Section 11 on page 357.

- **OUT**
  An argument of an MPI procedure call with the following property: the call may update the argument but does not use its input value. (See also IN and INOUT.) Refer to Section 2.4 on page 11.

- **pack**
  The process of copying data into a contiguous buffer. (See also unpack.) Refer to Section 4.2 on page 121.

- **passive target communication**
  An RMA communication where data is moved from the memory of one process to the memory of another, and only the origin process is explicitly involved in the transfer. Thus, two origin processes may communicate by accessing the same location in a target window. The process that owns the target window may be distinct from the two communicating processes, in which case it does not participate explicitly in the communication. This communication paradigm is closest to a shared memory model, where shared data can be accessed by all processes, irrespective of location. (See also active target communication and RMA.) Refer to Section 11.4 on page 369.

- **persistent communication**
  A type of communication that binds the list of communication arguments to a **persistent** communication request once and, then, repeatedly uses the request to initiate and complete messages. For example, this is useful when a communication pattern

with the same argument list is repeatedly executed within the inner loop of a parallel computation. The persistent request thus created can be thought of as a communication port or a "half-channel." It does not provide the full functionality of a conventional channel, since there is no binding of the send port to the receive port. This construct allows reduction of the overhead for communication between the process and communication controller, but not of the overhead for communication between one communication controller and another. It is not necessary that messages sent with a persistent request be received by a receive operation using a persistent request, or vice versa. Refer to Section 3.9 on page 69.

- **PMPI**
  Profiling MPI interface. A name-shift interface that provides a mechanism to analyze MPI function usage. Refer to Chapter 14 on page 475.

- **point-to-point communication**
  Messages delivered from one sending process to one receiving process. (See also collective.) Refer to Chapter 3 on page 25.

- **port name**
  A port_name is a *system-supplied* string that encodes a low-level network address at which a server can be contacted. Typically this is an IP address and a port number, but an implementation is free to use any protocol or naming convention. (See also service name, client and server). Refer to Section 10.4 on page 340.

- **portable datatype**
  A datatype is portable if it is a predefined datatype or it is derived from a portable datatype using only the type constructors MPI_TYPE_CONTIGUOUS, MPI_TYPE_VECTOR, MPI_TYPE_INDEXED, MPI_TYPE_CREATE_INDEXED_BLOCK, MPI_TYPE_CREATE_SUBARRAY, MPI_TYPE_DUP, and MPI_TYPE_CREATE_DARRAY. Such a datatype is portable because all displacements in the datatype are in terms of extents of one predefined datatype. Therefore, if such a datatype fits a data layout in one memory, it will fit the corresponding data layout in another memory, if the same declarations were used, even if the two systems have different architectures. On the other hand, if a datatype was constructed using MPI_TYPE_CREATE_HINDEXED or MPI_TYPE_CREATE_HVECTOR, then the datatype contains explicit byte displacements (e.g., providing padding to meet alignment restrictions). These displacements are unlikely to be chosen correctly if they fit data layout on one memory, but are used for data layouts on another process, running on a processor with a different architecture. Refer to Section 2.4 on page 11.

- **predefined datatype**
  (See also basic datatype and derived datatype.) Refer to Annex B on page 554.

- **process**
  An MPI program consists of autonomous processes, executing their own code, in an MIMD style. The codes executed by each process need not be identical. The processes communicate via calls to MPI communication primitives. Typically, each process executes in its own address space, although shared-memory implementations of MPI are possible. A process is represented in MPI by a (group, rank) pair. A (group, rank) pair specifies a unique process but a process does not determine a unique (group,

rank) pair, since a process may belong to several groups. Note that an MPI process may or may not be correlated with the operating system notion of a process. Refer to Section 2.6.5 on page 21.

- **process group**
  See groups. Refer to Section 3.2.3 on page 29, and Section 6 on page 209.

- **progress**
  The property of parallel and distributed systems that posted operations must complete in finite time unless an exceptional condition (such as resource exhaustion) causes an error. Refer to point-to-point progress in Section 3.6 on page 42, file I/O progress in Section 13.6.3 on page 463, and RMA progress in Section 11.7 on page 385).

- **ready communication mode**
  A point-to-point communication protocol in which the communication may be started *only* if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined. On some systems, this allows the removal of a handshake operation that is otherwise required and results in improved performance. The completion of the send operation does not depend on the status of a matching receive, and merely indicates that the send buffer can be reused. A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation; it is merely that the sender provides additional information to the system (namely that a matching receive is already posted), that can save some overhead. In a correct program, therefore, a ready send could be replaced by a standard send with no effect on the behavior of the program other than performance. (See also standard communication mode, buffered communication mode, synchronous communication mode.) Refer to Section 3.4 on page 38.

- **reduce operation**
  A collective procedure in which an operation is performed on data supplied by the members of a group. The reduction operation can be either one of a predefined list of operations or a user-defined operation. The global reduction functions come in several variations: a reduce that returns the result of the reduction to one member of a group, an all-reduce that returns this result to all members of a group, scan (parallel prefix) operations, non-blocking reductions, and certain RMA accumulate operations. Refer to Section 5.9 on page 162.

- **remote group**
  The destinations of a procedure for an intercommunicator. (See also intercommunicator and local group.) Refer to Section 5.2 on page 134.

- **type conversion**
  Changing the binary representation of a value, e.g., from Hex floating point to IEEE floating point. Note that the buffer size required for the receive can be affected by data conversions and by the stride and other datatype layout factors of the receive datatype. (See also representation conversion.) Refer to Section 3.3.2 on page 37.

- **RMA**
  Remote Memory Access (RMA) extends the communication mechanisms of MPI by allowing one process to specify all communication parameters, both for the sending

side and for the receiving side. In some systems, message-passing and remote-memory-access (RMA) operations run faster when accessing specially allocated memory (e.g., memory that is shared by the other processes in the communicating group on an SMP). MPI provides a mechanism for allocating and freeing such special memory. (See also active target, passive target, window, access epoch, exposure epoch, origin process, and target process.) Refer to Chapter 11 on page 357 and Section 8.2 on page 296).

- **root**
  The single process that originates or receives communication for those collective operations that originate or receive to one process (e.g., broadcast and gather). (See also collective communication.) Refer to Section 5.1 on page 131.

- **scan operation**
  A collective procedure in which a partial reduction operation is performed on data supplied by the members of a group. In the inclusive scan (see MPI_SCAN), the prefix reduction on process i includes the data from process i. (See also exscan operation and reduce operation.) Refer to Section 5.9 on page 162.

- **scope**
  The domain over which the service_name can be retrieved when establishing communication among two sets of MPI processes that do not share a communicator. [Refer to Section 10.4 on page 340.

- **send-receive operation**
  Operations that combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) may be the same. A send-receive operation is useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly (for example, even processes send, then receive, odd processes receive first, then send) so as to prevent cyclic dependencies that may lead to deadlock. When a send-receive operation is used, the communication subsystem takes care of these ordering issues. Refer to Section 3.10 on page 74.

- **sequential storage**
  Variables that belong to the same array, to the same COMMON block in Fortran, or to the same structure in C. Refer to Section 4.1.12 on page 104.

- **serialized**
  MPI calls are not made concurrently from two distinct threads in same process (all MPI calls are *serialized*). Refer to Section 12.4 on page 403.

- **server**
  The member of a client/server relationship that accepts requests. MPI provides a mechanism for two sets of MPI processes that do not share a communicator to establish communication. Establishing contact between two groups of processes that do not share an existing communicator is a collective but asymmetric process. One group of processes indicates its willingness to accept connections from other groups of processes. This group is called the (parallel) *server*, even if this is not a client/server type of

application. (See also client, port name, and service name.) Refer to Section 10.4 on page 340.

- **service name**
  A service_name is an *application-supplied* string for establishing communication between two independently running MPI applications. A service name can be published so that it can be used by a client to connect to a server without knowing the port_name. (See also port name). Refer to Section 10.4 on page 340.

- **split collective**
  begin discussion >————————
  this may be deprecated in mpi-3
  ———————— < **end discussion**
  A restricted form of "nonblocking" operations for collective file data access. These routines are referred to as "split" collective routines because a single collective operation is split in two: a begin routine and an end routine. The begin routine begins the operation, much like a nonblocking data access (e.g., MPI_FILE_IREAD). The end routine completes the operation, much like the matching test or wait (e.g., MPI_WAIT). As with nonblocking data access operations, the user must not use the buffer passed to a begin routine while the routine is outstanding; the operation must be completed with an end routine before it is safe to free buffers, etc. Refer to Section 13.4.5 on page 443.

- **standard communication mode**
  A point-to-point communication protocol that leaves it up to MPI to decide whether outgoing messages will be buffered. MPI may buffer outgoing messages. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete until a matching receive has been posted, and the data has been moved to the receiver. Thus, a send in standard mode can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. The standard mode send is **non-local**: successful completion of the send operation may depend on the occurrence of a matching receive. (See also buffered communication mode, synchronous communication mode, ready communication mode.) Refer to Section 3.4 on page 38.

- **synchronizing collective communication**
  Communication between MPI processes with the effect of constraining the relative temporal order that those MPI processes execute code. For example, MPI_BARRIER blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call. Refer to Section 5.1 on page 131.

- **synchronous communication mode**
  A point-to-point communication mode in which the communication can be started whether or not a matching receive was posted. However, the send will only complete successfully once a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but it also indicates that the receiver has reached a certain point in its execution, namely that

it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is non-local. (See also standard communication mode, buffered communication mode, ready communication mode.) Refer to Section 3.4 on page 38.

- **tag**
  An integer in point-to-point communication included as part of the *message envelope*. This integer can be used by the program to distinguish different types of messages. The range of valid tag values is 0,...,UB, where the value of UB is implementation dependent. (See also message envelope.) Refer to Section 3.2.3 on page 29.

- **target process**
  The process in which the memory is accessed in one-sided communications. (See also origin process and RMA.) Refer to Section 11 on page 357.

- **thread compliant implementation**
  An MPI process that may be multithreaded. (See also thread safe and serialized.) Refer to Section 12.4 on page 403.

- **thread safe**
  The property that two concurrently running threads may make MPI calls and the outcome will be as if the calls executed in some order, even if their execution is interleaved. (See also thread compliant implementation.) Refer to Section 12.4 on page 403.

- **topology**
  The layout of processes within a virtual structure such as two or three dimensional grid. A topology is an extra, optional attribute that one can give to an intra-communicator; topologies cannot be added to inter-communicators. A topology can provide a convenient naming mechanism for the processes of a group (within a communicator), and additionally, may assist the runtime system in mapping the processes onto hardware. (See also groups, cartesian topology, and graph topology.) Refer to Section 7 on page 267.

- **true extent**
  The true size of a datatype, i.e., the extent of the corresponding typemap, ignoring MPI_LB and MPI_UB markers, and performing no rounding for alignment. The true_extent is the minimum number of bytes of memory necessary to hold a datatype. If the typemap associated with datatype is

$$Typemap = \{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$$

  Then

$$true\_lb(Typemap) = min_j\{disp_j \ : \ type_j \neq \mathbf{lb}, \mathbf{ub}\},$$

$$true\_ub(Typemap) = max_j\{disp_j + sizeof(type_j) \ : \ type_j \neq \mathbf{lb}, \mathbf{ub}\},$$

and

$$true\_extent(Typemap) = true\_ub(Typemap) - true\_lb(typemap).$$

(See also lower bound datatype, upper bound datatype, and extent.) Refer to Section 4.1 on page 77.

- **type conversion**
  Changing the datatype of a value, e.g., by rounding a `REAL` to an `INTEGER`. (See also representation conversion.) Refer to Section 3.3.2 on page 37.

- **type map**
  The pair of sequences (or sequence of pairs) associated with a general dataype. The displacements are not required to be positive, distinct, or in increasing order. Therefore, the order of items need not coincide with their order in store, and an item may appear more than once. Type maps take the form

  $$Typemap = \{(type_0, disp_0), ..., (type_{n-1}, disp_{n-1})\},$$

  where $type_i$ are basic types, and $disp_i$ are displacements. (See also type signature, basic datatype, and derived datatype.) Refer to Section 4.1 on page 77.

- **type signature**
  The sequences of types associated with a general dataype. Type signatures may be used to validate matching types between sender and receiver; they take the form

  $$Typesig = \{type_0, ..., type_{n-1}\}$$

  where $type_i$ are basic datatypes. (See also type map and basic datatype.) Refer to Section 4.1 on page 77.

- **unpack**
  The process of copying a contiguous buffer to a second buffer according to an MPI datatype. Note that an MPI datatype may specify a non-contiguous structure. (See also pack.) Refer to Section 4.2 on page 121.

- **upper bound, datatype**
  The displacement of the highest unit of store which is addressed by the datatype. (See also lower bound and extent.) Refer to ub(Typemap) in Section 4.9 on page 96.

- **view, file**
  A *view* defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view. The pattern of repetition is defined to be the same pattern that MPI_TYPE_CONTIGUOUS would produce if it were passed the filetype and an arbitrarily large count. Refer to Section 13.1.1 on page 411.

- **wildcard**
  The receive of a point-to-point message may utilize a special *tag* (MPI_ANY_TAG) or *source* (MPI_ANY_SOURCE) that indicates any source and/or tag are acceptable. Note that wildcards may not be used to constrain *communicators*. Refer to Section 11 on page 357.

- **window**
  A range of memory that is made accessible to accesses by remote processes using one sided communication. In one-sided communications, each process specifies a window of existing memory that it exposes to RMA accesses by the processes in the group of comm. The window consists of size bytes, starting at address base. (See also RMA.) Refer to Section 11 on page 357.