Annex A

Glossary

- absolute address
  Displacements relative to "address zero," the start of the address space. See Section ?? on page ??.
- access epic
  The period during which a target window can be accessed by **RMA** operations. (See also ??.) See Section 11.4 on page 369.
- active
  We say that a parallel procedure is *active* in a process if the process 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. See Section 6.9 on page 264.
- active target
  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 communication. 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 A.) See Section 11.4 on page 369.
- type associative
  The property of a collective reduction, namely that the order in which the operations are performed does not matter as long as the sequence of the operands is not changed. (See also A.) See Section 5.9.5 on page 171.
- blocking
  A procedure is blocking if return from the procedure indicates the user is allowed to reuse resources specified in the call.
- local] A procedure is local if completion of the procedure depends only on the local executing process. See Section 2.4 on page 11.
- broadcast
  A collective operation which communicates data from a root process to all

processes; initially just the first process contains the data, but after the broadcast all processes contain it. See Section 5.1 on page 131.

- buffered communication mode
  A 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 A, A, A.) See Section 3.4 on page 38.

- caching
  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 a portable 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 Section 6.7 on page 246.

- client
  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 connects to the server; we will call it the *client*. (See also A). See Section 10.4 on page 340.

- collective
  A procedure is collective if 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 Section 2.4 on page 11.

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

- communicator
  A communicator specifies the communication context for a communication

operation. Each communication context provides a separate "communication universe:" messages are always received within the context they were sent, and messages sent in different contexts do not interfere. The communicator also specifies the set of processes that share this communication context. See Section 3.2.3 on page 29.

- communication context
  See A.

- type commutative
  The property of a collective reduction, namely that changing the order of the operands does not change the end result. (See also A.) See Section 5.9.5 on page 171.

- type 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 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 non persistent, it is freed, and becomes inactive if it was persistent. A communication completes when all participating operations complete. See Section 2.4 on page 11.

- contexts
  Contexts provide the ability to have a separate safe "universe" of message-passing between the two groups. A send in the local group is always a receive in the remote group, and vice versa. 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. See Section 6 on page 209.

- contiguous
  A collection of memory locations that are adjacent to one another without intervening extraneous data. See Section 5.1 on page 131.

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

- created datatype
  The initial step in preparing a datatype for communication (before "committed datatype"). (See also A.) See 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 MPI-2. 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 INTEGERs and 64 bit addresses. See Section 2.6.1 on page 16.

- **derived datatype**
  A derived datatype is any datatype that is not predefined. (See also A and A .) See Section 2.4 on page 11.

- **displacement**
  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." See Section 13.1.1 on page 411.

- **equivalent**
  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. See Section 2.4 on page 11.

- **erroneous program**
  A program that does not perform as intended; A program that contains bugs. (See also A.) See Section 5.13 on page 200.

- **error class**
  The error codes returned by MPI are left entirely to the implementation (with the exception of MPI_SUCCESS). 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. See Section 8.1 on page 306.

- **etype**
  An *etype* (*elementary* datatype) is the unit of data access and positioning. 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. See Section 13.1.1 on page 411.

- **exposure epic**
  The period during which a target window can be accessed by RMA operations in *active target* communication. (See also ??.) See 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

$$\begin{aligned} lb(Typemap) &= \min_j disp_j, \\ ub(Typemap) &= \max_j (disp_j + sizeof(type_j)) + \epsilon, \text{ and} \\ extent(Typemap) &= ub(Typemap) - lb(Typemap). \end{aligned} \tag{A.1}$$

  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 A, A, and A.) See Section 4.1 on page 77.

- **external32**
  Data in *external32* data representation states that read and write operations convert all data from and to the "external32" representation defined in Section 13.5.2, page 453. The data conversion rules for communication also apply to these conversions (see Section 3.3.2, page 25-27, of the MPI-1 document). The data on the storage medium is always in this canonical representation, and the data in memory is always in the local process's native representation. (See also A and A.) See 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, yet the message is never received, because it is each time overtaken by another message, sent from another source. Similarly, suppose that a receive was posted by a multithreaded 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 node (by other executing threads). It is the programmer's responsibility to prevent starvation in such situations. See Section 3.7 on page 43.

- 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. See 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. See Section ?? on page ??.

- file handle
  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. See Section 13.1.1 on page 411.

- file interoperability
  file interoperability is the ability to read the information previously written to a file—not just the bits of data, but the actual information the bits represent. MPI guarantees full interoperability within a single MPI environment, and supports increased interoperability outside that environment through the external data representation (Section 13.5.2, page 453) as well as the data conversion functions (Section 13.5.3, page 454). See 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. See 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. Using the size as an absolute displacement gives the position of the byte immediately following the last byte in the file. For any given view, the *end of file* is the offset of the first

etype accessible in the current view starting after the last byte in the file. See Section 13.1.1 on page 411.

- 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, See Section 13.1.1 on page 411.

- freed datatype
  To mark a datatype object associated with **datatype** for deallocation and set the **datatype** to **MPI_DATATYPE_NULL**. Any communication that is currently using this datatype will complete normally. See Section 4.1.9 on page 99.

- gather
  A collective operation in which `n` messages sent by the processes in the group 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, ...)**. See Section 5.1 on page 131.

- general datatype
  A general datatype is an opaque object 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 A and A .) See Section 2.4 on page 11.

- generalized request
  A user defined non-blocking operation. See Section 12.1 on page 395.

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

- groups
  Groups define an ordered collection of processes, each with a rank, and it is this group that defines the low-level names for inter-process communication (ranks are used for sending and receiving). Thus, groups define a scope for process names in point-to-point communication. In addition, groups define the scope of collective operations. Groups may be manipulated separately from communicators in **MPI**, but only communicators can be used in communication operations. Each process in the group is assigned a rank between `0` and `n-1`. (See also A .) See Section 6 on page 209.

- implementation
  A specific fulfillment of a specification. See 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 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 Section 2.4 on page 11.

- **in place**
  A collective communication in which the output buffer is identical to the input 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. See Section 5.2 on page ??.

- **intercommunicator**
  A communicator that identifies two distinct groups of processes linked with a context. (See also A.) See Section 5.2 on page 134.

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

- **internal**
  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 A and A.) See Section 13.5 on page 450.

- **intracommunicator**
  ticket109. **A communicator that can be thought of as an i[n]dentifier for a single group of processes linked with a context. (See also A.) See Section 5.2 on page 134.**

- **lower bound**
  The displacement of the lowest unit of store which is addressed by the datatype. In general, if

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

  then the lower bound of $Typemap$ is defined to be

$$lb(Typemap) = \begin{cases} \min_j disp_j & \text{if no entry has basic type lb} \\ \min_j \{disp_j \text{ such that } type_j = \mathsf{lb}\} & \text{otherwise} \end{cases}$$

  (See also A and A.) See 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 the communicator is identical, and the source rank and tag match, i.e., are identical or wildcarding isused. See Section 3.3.1 on page 34.

- message envelope
  In addition to the data part, messages carry information that can be used to distinguish messages and selectively receive them. This information consists of a fixed number of fields, which we collectively call the message envelope. These fields are

  source
  destination
  tag
  communicator

  See Section 3.2.3 on page 29.

- native
  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 A and A.) See Section 13.5 on page 450.

- non-local
  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. See 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 this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. See Section 2.4 on page 11.

- nonblocking
  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. See Section 3.5 on page 42.

- 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. See Section 8.5 on page 308.

- null handle
  A handle with value **MPI_REQUEST_NULL**. See Section 3.7.3 on page 53.

- null process
  A "dummy" source or destination 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. See Section 3.11 on page 76.

- 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 A.2 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. See Section 13.1.1 on page 411.

- opaque
  Data objects managed by MPI whose size and shape are not visible to the user. These objects are said to reside in system space. Opaque objects are accessed by the application program through handles, e.g., a communicator handles. See Section 2.5.1 on page 12.

- order
  The requirement that operations be performed as the sequence of the calls that initiate the communication. See Section 3.7.4 on page 56.

- origin
  The process that performs the call in on-sided communications. See 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 Section 2.4 on page 11.

- **pack**
  The process of copying data into a contiguous buffer before sending it. (See also A.) See Section 4.2 on page 121.

- **passive target**
  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 A.) See Section 11.4 on page 369.

- **persistent communication request**
  An optimization available when a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a persistent communication request once and, then, repeatedly using the request to initiate and complete messages. 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.. See Section 3.9 on page 69.

- **PMPI**
  Profiling MPI interface. A name-shift interface that provides a mechanism to analyze MPI function usage. See Section 16.1.10 on page 499.

- **point-to-point**
  Messages delivered from one sending process to one receiving process. (See also A.) See Section 3.1 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. See Section 10.4 on page 340.

- portable
  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, MPI_TYPE_CREATE_HVECTOR or MPI_TYPE_CREATE_STRUCT, 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. See Section 2.4 on page 11.

- predefined datatype
  A predefined datatype is a datatype with a predefined (constant) name (such as MPI_INT, MPI_FLOAT_INT, or MPI_UB) or a datatype constructed with MPI_TYPE_CREATE_F90_INTEGER, MPI_TYPE_CREATE_F90_REAL, or MPI_TYPE_CREATE_F90_COMPLEX. The former are named whereas the latter are unnamed. (See also A and A .) See Section 2.4 on page 11.

- 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. See Section 2.6.5 on page 21.

- process group
  An ordered list of processes that share a communicator context. Processes are identified by their rank within this group. Thus, the range of valid values for **dest** is 0, ... , n-1, where n is the number of processes in the group. See Section 3.2.3 on page 29.

- progress
  The requirement that if a pair of matching send and receives have been initiated on two processes, then at least one of these two operations will complete, independently of other actions in the system: the send operation will complete, unless the receive is satisfied by another message, and completes; the receive operation will complete, unless the message sent is consumed by another matching receive that was posted at the same

destination process. See Section 3.6 on page 42.

- ready communication mode
  A 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 hand-shake 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 A, A, A.) See Section 3.4 on page 38.

- reduce
  A collective operation that determines a result for all 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 flavors: 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, and two scan (parallel prefix) operations. See Section 5.9 on page 162.

- 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 of the receive datatype. No conversion need occur when an MPI program executes in a homogeneous system, where all processes run in the same environment. (See also A.) See Section 3.3.2 on page 37.

- RMA
  Remote Memory Access. 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. The use of such memory for message-passing or RMA is not mandatory, and this memory can be used without restrictions as any other dynamically allocated memory. (See also A and A.) See 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 Section 5.1 on page 131.

- scope
  The domain over which the service_name can be retrieved. [See 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) are possibly the same. A send-receive operation is very 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 issues. See 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. See Section 4.1.12 on page 104.

- sequential storage
  MPI calls are not made concurrently from two distinct threads (all MPI calls are *serialized*. See Section 12.4 on page 403.

- 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. One group of processes indicates its willingness to accept connections from other groups of processes. We will call this group the (parallel) *server*, even if this is not a client/server type of application. (See also A). See Section 10.4 on page 340.

- split collective
  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. See Section 13.4.5 on page 443.

- standard communication mode
  A 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 A, A, A.) See Section 3.4 on page 38.

- synchronizing
  Communication between MPI processes with the effect of constraining the relative 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. See Section 5.1 on page 131.

- synchronous communication mode
  A communication protocol in which the communication can be started whether or not a matching receive was posted. However, the send will complete successfully only if 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 A, A, A.) See Section 3.4 on page 38.

- target
  The process in which the memory is accessed in on-sided communications. See Section 11 on page 357.

- 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 Section 5.1 on page 131.

- topology
  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 run-time system in mapping the processes onto hardware. (See also A .) See 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, uncompressed. (Note that this applies to

situations such as spanning trees; since the receive buffer is only valid on the root process, one will need to allocate some temporary space for receiving data on intermediate nodes. However, the datatype extent cannot be used as an estimate of the amount of space that needs to be allocated, if the user has modified the extent using the MPI_UB and MPI_LB values.) 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 A, A, and A.) See Section 4.1 on page 77.

- **type conversion**
  Changing the datatype of a value, e.g., by rounding a `REAL` to an `INTEGER`. Note that the buffer size required for the receive can be affected by data conversions and by the stride of the receive datatype. No conversion need occur when an MPI program executes in a homogeneous system, where all processes run in the same environment. (See also A.) See Section 3.3.2 on page 37.

- **type map**
  The pair of sequences (or sequence of pairs) associated with a general datatype. 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

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

  be such a type map, where $type_i$ are basic types, and $disp_i$ are displacements. (See also A.) See Section 4.1 on page 77.

- **type signature**
  The sequences of types associated with a general datatype. 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 types. (See also A.) See Section 4.1 on page 77.

- **unpack**
  The process of copying data into a contiguous buffer after receiving it. (See also A.) See Section 4.2 on page 121.

- **upper bound**
  The displacement of the highest unit of store which is addressed by the datatype. In general, if

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

  then the upper bound of $Typemap$ **is defined to be**

$$ub(Typemap) = \begin{cases} \max_j disp_j + sizeof(type_j) + \epsilon & \text{if no entry has basic type ub} \\ \max_j\{disp_j \text{ such that } type_j = \mathsf{ub}\} & \text{otherwise} \end{cases}$$

  (See also A and A.) See Section 4.9 on page 96.

- **view**
  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. Figure ?? shows how the tiling works; note that the filetype in this example must have explicit lower and upper bounds set in order for the initial and final holes to be repeated in the view. Views can be changed by the user during program execution. The default view is a linear byte stream (displacement is zero, etype and filetype equal to MPI_BYTE).**

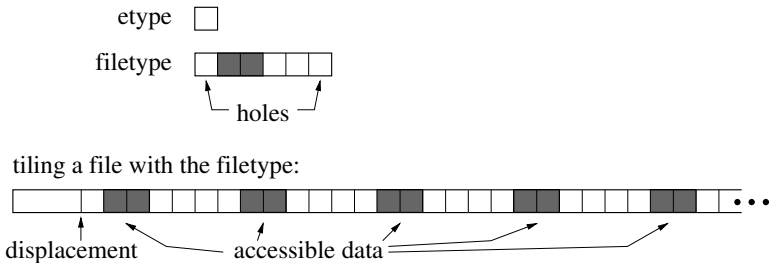

Figure A.1: Etypes and filetypes

A group of processes can use complementary views to achieve a global data distribution such as a scatter/gather pattern (see Figure A.2).

See Section 13.1.1 on page 411.

- **wildcard**
  A special tag that will match all messages. Wildcard values may be used to accept all message *sources* **and/or** *tags***, but may not be used to constrain** *communicators***. See Section 11 on page 357.**

Figure A.2: Partitioning a file among parallel processes

- **window**
  A range of memory that is made accessible to accesses by remote processes. 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 Section 3.5 on page 42.