1 Programming Using the Message-Passing Paradigm

- A message passing architecture uses a set of primitives that allows processes to communicate with each other. These include the *send*, *receive*, *broadcast*, and *barrier* primitives.
- Numerous programming languages and libraries have been developed for explicit parallel programming. These differ in their view of the address space that they make available to the programmer, the degree of synchronization imposed on concurrent activities, and the multiplicity of programs.

2 Principles of Message-Passing Programming

- There are two key attributes that characterize the message-passing programming paradigm.
  1. the first is that it assumes a partitioned address space
  2. the second is that it supports only explicit parallelization.
- There are two immediate implications of a partitioned address space.
  - First, each data element must belong to one of the partitions of the space; hence, data must be explicitly partitioned and placed. This adds complexity to programming, but encourages locality of access that is critical for achieving high performance on non-UMA architecture, since a processor can access its local data much faster than non-local data on such architectures.
  - The second implication is that all interactions (read-only or read/write) require cooperation of two processes (the process that has the data and the process that wants to access the data). This requirement for cooperation adds a great deal of complexity for a number of reasons.
- The process that has the data must participate in the interaction even if it has no logical connection to the events at the requesting process. In certain circumstances, this requirement leads to unnatural programs. In particular, for dynamic and/or unstructured interactions the complexity of the code written for this type of paradigm can be very high for this reason.

- However, a primary advantage of explicit two-way interactions is that the programmer is fully aware of all the costs of non-local interactions, and is more likely to think about algorithms (and mappings) that minimize interactions.
- Another major advantage of this type of programming paradigm is that it can be efficiently implemented on a wide variety of architectures.
- The programmer is responsible for analyzing the underlying serial algorithm/application and identifying ways by which he or she can decompose the computations and extract concurrency.
- As a result, programming using the message-passing paradigm tends to be hard and intellectually demanding. However, on the other hand, properly written message-passing programs can often achieve very high performance and scale to a very large number of processes.

2.0.1 Structure of Message-Passing Programs

- Message-passing programs are often written using the *asynchronous* or *loosely synchronous* paradigms.
  - In the *asynchronous* paradigm, all concurrent tasks execute asynchronously. This makes it possible to implement any parallel algorithm. However, such programs can be harder to reason about, and can have non-deterministic behavior due to race conditions.
  - *Loosely synchronous* programs are a good compromise between these two extremes. In such programs, tasks or subsets of tasks synchronize to perform interactions. However, between these interactions, tasks execute completely asynchronously. Since the interaction happens synchronously, it is still quite easy to reason about the program. Many of the known parallel algorithms can be naturally implemented using loosely synchronous programs.
- In its most general form, the message-passing paradigm supports execution of a different program on each of the p processes. This provides the ultimate flexibility in parallel programming, but makes the job of writing parallel programs effectively unscalable.
- For this reason, most message-passing programs are written using the single program multiple data (SPMD) approach. In SPMD programs the code executed by different processes is identical except for a small number of processes (e.g., the ”root” process).

- This does not mean that the processes work in lock-step. In an extreme case, even in an SPMD program, each process could execute a different code (the program contains a large case statement with code for each process). But except for this degenerate case, most processes execute the same code. SPMD programs can be loosely synchronous or completely asynchronous.

2.1 The Building Blocks: Send and Receive Operations

- Since interactions are accomplished by *sending* and *receiving* messages, the basic operations in the message-passing programming paradigm are **send** and **receive**. In their simplest form, the prototypes of these operations are defined as follows:

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

  - *sendbuf* points to a buffer that stores the data to be sent,
  - *recvbuf* points to a buffer that stores the data to be received,
  - *nelems* is the number of data units to be sent and received,
  - *dest* is the identifier of the process that receives the data,
  - *source* is the identifier of the process that sends the data.

```
      P0                               P1

      a = 100;                         receive(&a, 1, 0)
      send(&a, 1, 1);                  printf("%d\n", a);
      a=0;
```

- Process P_0 sends a message to process P_1 which receives and prints the message.
- The important thing to note is that process P_0 changes the value of a to 0 immediately following the send. The semantics of the send operation require that the value received by process P_1 must be 100 as opposed to 0. That is, the value of a at the time of the send operation must be the value that is received by process P_1.

- It may seem that it is quite straightforward to ensure the semantics of the send and receive operations. However, based on how the send and receive operations are implemented this may not be the case.

- Most message passing platforms have additional hardware support for sending and receiving messages. They may support DMA (direct memory access) and asynchronous message transfer using network interface hardware.

- Network interfaces allow the transfer of messages from buffer memory to desired location without CPU intervention. Similarly, DMA allows copying of data from one memory location to another (e.g., communication buffers) without CPU support (once they have been programmed).

- As a result, if the send operation programs the communication hardware and returns before the communication operation has been accomplished, process P_1 might receive the value 0 in a instead of 100!

2.1.1 Blocking Message Passing Operations

- A simple solution to the dilemma presented in the code fragment above is for the send operation to return only when it is semantically safe to do so.

- Note that this is not the same as saying that the send operation returns only after the receiver has received the data. It simply means that the sending operation blocks until it can guarantee that the semantics will not be violated on return irrespective of what happens in the program subsequently. There are two mechanisms by which this can be achieved.

1. Blocking Non-Buffered Send/Receive

    - In the first case, the send operation does not return until the matching receive has been encountered at the receiving process. When this happens, the message is sent and the send operation returns upon completion of the communication operation.

    - Typically, this process involves a handshake between the sending and receiving processes. The sending process sends a request to communicate to the receiving process. When the receiving process encounters the target receive, it responds to the request. The sending process upon receiving this response initiates a transfer operation (see Fig. 1). Since there are no buffers used at either

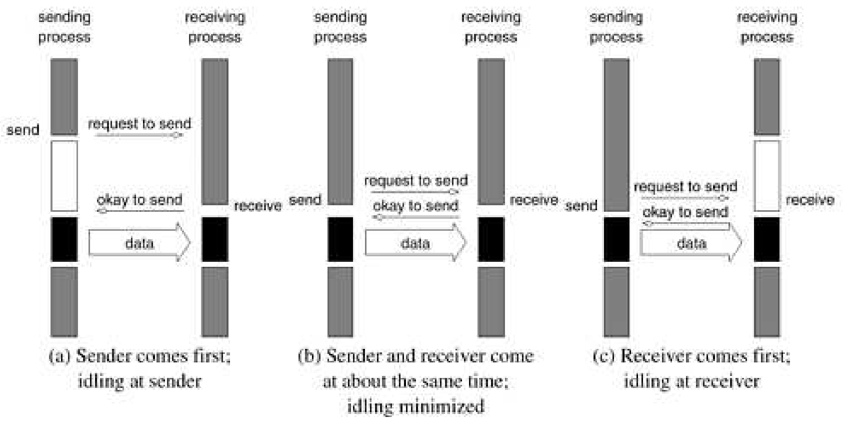
sending process
receiving process
send
request to send
okay to send
receive
data
(a) Sender comes first; idling at sender
(b) Sender and receiver come at about the same time; idling minimized
(c) Receiver comes first; idling at receiver

Figure 1: Handshake for a blocking non-buffered send/receive operation.

sending or receiving ends, this is also referred to as a non-buffered blocking operation.

- *Idling Overheads in Blocking Non-Buffered Operations:* It is clear from the figure that a blocking non-buffered protocol is suitable when the send and receive are posted at roughly the same time. However, in an asynchronous environment, this may be impossible to predict. This idling overhead is one of the major drawbacks of this protocol.
- *Deadlocks in Blocking Non-Buffered Operations:* Consider the following simple exchange of messages that can lead to a deadlock:

```
    P0                                   P1

send(&a, 1, 1);                     send(&a, 1, 0);
receive(&b, 1, 1);                  receive(&b, 1, 0);
```

  The code fragment makes the values of a available to both processes P_0 and P_1. However, if the send and receive operations are implemented using a blocking non-buffered protocol, the send at P_0 waits for the matching receive at P_1 whereas the send at process P_1 waits for the corresponding receive at P_0, resulting in an infinite wait. Deadlocks are very easy in blocking protocols and care must be taken to break cyclic waits of the nature outlined.

2. Blocking Buffered Send/Receive

- A simple solution to the *idling* and *deadlocking* problem outlined above is to rely on buffers at the sending and receiving ends.

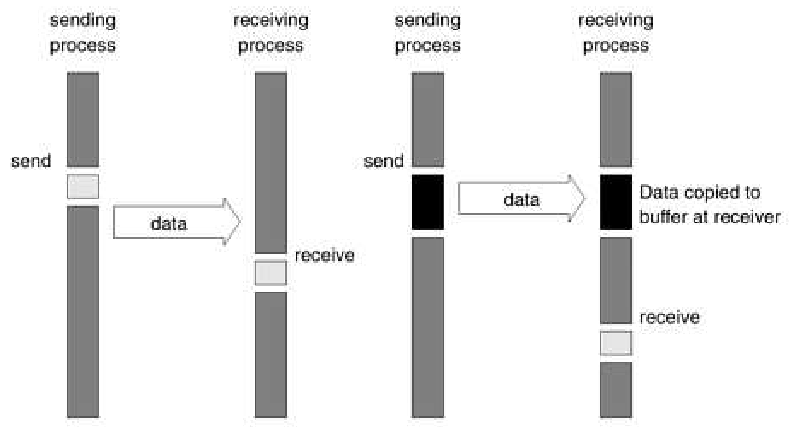

Figure 2: Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

- Figure 2a
  - On a send operation, the sender simply copies the data into the designated buffer and returns after the copy operation has been completed.
  - The sender process can now continue with the program knowing that any changes to the data will not impact program semantics.
  - The actual communication can be accomplished in many ways depending on the available hardware resources. If the hardware supports asynchronous communication (independent of the CPU), then a network transfer can be initiated after the message has been copied into the buffer.
  - Note that at the receiving end, the data cannot be stored directly at the target location since this would violate program semantics. Instead, the data is copied into a buffer at the receiver as well.
  - When the receiving process encounters a receive operation, it checks to see if the message is available in its receive buffer. If so, the data is copied into the target location.
- Figure 2b

  - In the protocol illustrated Fig. 2a, buffers are used at both sender and receiver and communication is handled by dedicated hardware. Sometimes machines do not have such communication hardware.
  - In this case, some of the overhead can be saved by buffering only on one side. For example, on encountering a send operation, the sender interrupts the receiver, both processes participate in a communication operation and the message is deposited in a buffer at the receiver end.
  - When the receiver eventually encounters a receive operation, the message is copied from the buffer into the target location.

- It is easy to see that buffered protocols alleviate idling overheads at the cost of adding buffer management overheads.
- In general, if the parallel program is highly synchronous (i.e., sends and receives are posted around the same time), non-buffered sends may perform better than buffered sends. However, in general applications, this is not the case and buffered sends are desirable unless buffer capacity becomes an issue.
- Impact of finite buffers in message passing; consider the following code fragment:

```
   P0                                      P1

for (i = 0; i < 1000; i++) {          for (i = 0; i < 1000; i++) {
 produce_data(&a);                     receive(&a, 1, 0);
 send(&a, 1, 1);                       consume_data(&a);
}                                     }
```

  - In this code fragment, process P_0 produces 1000 data items and process P_1 consumes them. However, if process P_1 was slow getting to this loop, process P_0 might have sent all of its data.
  - If there is enough buffer space, then both processes can proceed; however, if the buffer is not sufficient (i.e., buffer overflow), the sender would have to be blocked until some of the corresponding receive operations had been posted, thus freeing up buffer space.
  - This can often lead to unforeseen overheads and performance degradation. In general, it is a good idea to write programs that have bounded buffer requirements.

- *Deadlocks in Buffered Send and Receive Operations:* While buffering alleviates many of the deadlock situations, it is still possible to write code that deadlocks. This is due to the fact that as in the non-buffered case, receive calls are always blocking (to ensure semantic consistency). Thus, a simple code fragment such as the following deadlocks since both processes wait to receive data but nobody sends it.

  ```
    P0                        P1

  receive(&a, 1, 1);        receive(&a, 1, 0);
  send(&b, 1, 1);           send(&b, 1, 0);
  ```

  Once again, such circular waits have to be broken. However, deadlocks are caused only by waits on receive operations in this case.

2.1.2 Non-Blocking Message Passing Operations

- In blocking protocols, the overhead of guaranteeing semantic correctness was paid in the form of idling (non-buffered) or buffer management (buffered).
- Often, it is possible to require the programmer to ensure semantic correctness and provide a fast send/receive operation that incurs little overhead.
- This class of non-blocking protocols returns from the send or receive operation before it is semantically safe to do so. Consequently, the user must be careful not to alter data that may be potentially participating in a communication operation.
- Non-blocking operations are generally accompanied by a check-status operation, which indicates whether the semantics of a previously initiated transfer may be violated or not.
- Upon return from a non-blocking send or receive operation, the process is free to perform any computation that does not depend upon the completion of the operation. Later in the program, the process can check whether or not the non-blocking operation has completed, and, if necessary, wait for its completion.
- As illustrated in Fig. 3, non-blocking operations can themselves be buffered or non-buffered.

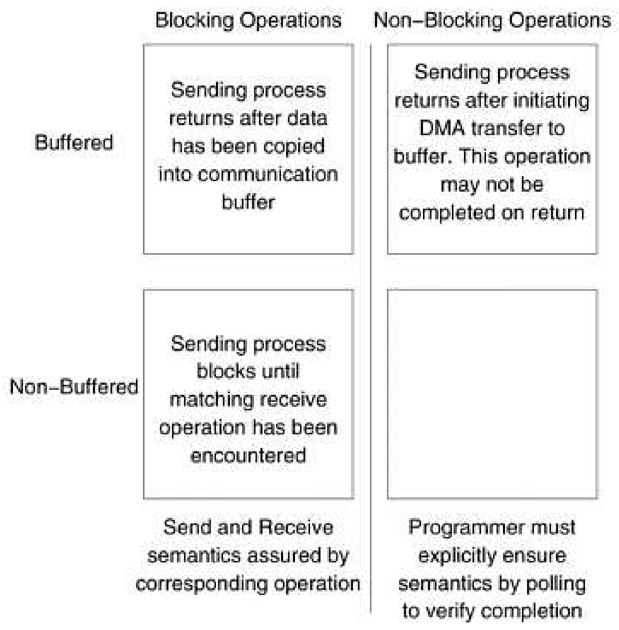
Blocking Operations
Non–Blocking Operations
Buffered
Non–Buffered
Sending process returns after data has been copied into communication buffer
Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return
Sending process blocks until matching receive operation has been encountered
Send and Receive semantics assured by corresponding operation
Programmer must explicitly ensure semantics by polling to verify completion

Figure 3: Space of possible protocols for send and receive operations.

- In the non-buffered case, a process wishing to send data to another simply posts a pending message and returns to the user program. The program can then do other useful work. At some point in the future, when the corresponding receive is posted, the communication operation is initiated.
- When this operation is completed, the check-status operation indicates that it is safe for the programmer to touch this data. This transfer is indicated in Fig. 4a.
- Comparing Figures 4a and 1a, it is easy to see that the idling time when the process is waiting for the corresponding receive in a blocking operation can now be utilized for computation, provided it does not update the data being sent.
- This alleviates the major bottleneck associated with the former at the expense of some program restructuring. The benefits of non-blocking operations are further enhanced by the presence of dedicated communication hardware.
- This is illustrated in Fig. 4b. In this case, the communication overhead can be almost entirely masked by non-blocking operations. In this case,

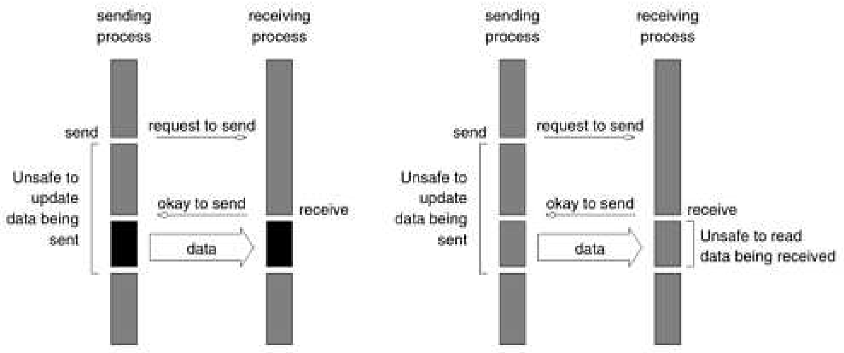

Figure 4: Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.

however, the data being received is unsafe for the duration of the receive operation.

- Non-blocking operations can also be used with a buffered protocol. In this case, the sender initiates a DMA operation and returns immediately.

- The data becomes safe the moment the DMA operation has been completed. At the receiving end, the receive operation initiates a transfer from the sender's buffer to the receiver's target location. Using buffers with non-blocking operation has the effect of reducing the time during which the data is unsafe.

- Typical message-passing libraries such as Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) implement both blocking and non-blocking operations.
  - Blocking operations facilitate safe and easier programming and non-blocking operations are useful for performance optimization by masking communication overhead.
  - One must, however, be careful using non-blocking protocols since errors can result from unsafe access to data that is in the process of being communicated.

Table 1: The minimal set of MPI routines.

MPI_Init	Initializes MPI
MPI_Finalize	Terminates MPI
MPI_Comm_size	Determines the number of processes
MPI_Comm_rank	Determines the label of the calling process
MPI_Send	Sends a message
MPI_Recv	Receives a message

2.2 MPI: the Message Passing Interface

- Many early generation commercial parallel computers were based on the message-passing architecture due to its lower cost relative to shared-address-space architectures.
- Message-passing became the modern-age form of assembly language, in which every hardware vendor provided its own library, that performed very well on its own hardware, but was incompatible with the parallel computers offered by other vendors.
- Many of the differences between the various vendor-specific message-passing libraries were only syntactic; however, often enough there were some serious semantic differences that required significant re-engineering to port a message-passing program from one library to another.
- The message-passing interface, or MPI as it is commonly known, was created to essentially solve this problem. MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran. The MPI standard defines both the syntax as well as the semantics of a core set of library routines that are very useful in writing message-passing programs.
- The MPI library contains over 125 routines, but the number of key concepts is much smaller. In fact, it is possible to write fully-functional message-passing programs by using only the six routines (see table 1).

2.2.1 Starting and Terminating the MPI Library

- MPI_Init is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment. Calling MPI_Init more than once during the execution of a program will lead to an error.

- MPI_Finalize is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment. No MPI calls may be performed after MPI_Finalize has been called, not even MPI_Init.
- Upon successful execution, MPI_Init and MPI_Finalize return MPI_SUCCESS; otherwise they return an implementation-defined error code.

2.2.2 Communicators

- A key concept used throughout MPI is that of the communication domain. A communication domain is a set of processes that are allowed to communicate with each other.
- Information about communication domains is stored in variables of type MPI_Comm, that are called *communicators*. These communicators are used as arguments to all message transfer MPI routines and they uniquely identify the processes participating in the message transfer operation.
- In general, all the processes may need to communicate with each other. For this reason, MPI defines a default communicator called MPI_COMM_WORLD which includes all the processes involved in the parallel execution.
- However, in many cases we want to perform communication only within (possibly overlapping) groups of processes. By using a different communicator for each such group, we can ensure that no messages will ever interfere with messages destined to any other group.

2.2.3 Getting Information

- The MPI_Comm_size and MPI_Comm_rank functions are used to determine the number of processes and the label of the calling process, respectively. The calling sequences of these routines are as follows:

  ```
  int MPI_Comm_size(MPI_Comm comm, int *size)
  int MPI_Comm_rank(MPI_Comm comm, int *rank)
  ```

- The function MPI_Comm_size returns in the variable size the number of processes that belong to the communicator *comm*.
- So, when there is a single process per processor, the call MPI_Comm_size(MPI_COMM_WORLD, &size) will return in *size* the number of processors used by the program.

- Every process that belongs to a communicator is uniquely identified by its *rank*. The rank of a process is an integer that ranges from zero up to the size of the communicator minus one.

- A process can determine its rank in a communicator by using the MPI_Comm_rank function that takes two arguments: the communicator and an integer variable rank. Up on return, the variable *rank* stores the rank of the process. Note that each process that calls either one of these functions must belong in the supplied communicator, otherwise an error will occur.

2.2.4 Sending and Receiving Messages

- The basic functions for sending and receiving messages in MPI are the MPI_Send and MPI_Recv, respectively. The calling sequences of these routines are as follows:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- MPI_Send sends the data stored in the buffer pointed by *buf*. This buffer consists of consecutive entries of the type specified by the parameter datatype.

- The number of entries in the buffer is given by the parameter *count*.

- The correspondence between MPI datatypes and those provided by C is shown in Table 2. Note that for all C datatypes, an equivalent MPI datatype is provided. However, MPI allows two additional datatypes that are not part of the C language. These are MPI_BYTE and MPI_PACKED.

  - MPI_BYTE corresponds to a byte (8 bits) and MPI_PACKED corresponds to a collection of data items that has been created by packing non-contiguous data.
  - Note that the length of the message in MPI_Send, as well as in other MPI routines, is specified in terms of the number of entries being sent and not in terms of the number of bytes.

Table 2: Correspondence between the datatypes supported by MPI and those supported by C.

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

  – Specifying the length in terms of the number of entries has the advantage of making the MPI code *portable*, since the number of bytes used to store various datatypes can be different for different architectures.

- The destination of the message sent by MPI_Send is uniquely specified by the *dest* and *comm* arguments.

- The *dest* argument is the *rank* of the destination process in the communication domain specified by the communicator *comm*.

- Each message has an integer-valued *tag* associated with it. This is used to distinguish different types of messages. The message-tag can take values ranging from zero up to the MPI defined constant MPI_TAG_UB. Even though the value of MPI_TAG_UB is implementation specific, it is at least 32,767.

- MPI_Recv receives a message sent by a process whose *rank* is given by the *source* in the communication domain specified by the *comm* argument.

- The *tag* of the sent message must be that specified by the tag argument.

If there are many messages with identical tag from the same process, then any one of these messages is received.

- MPI allows specification of wildcard arguments for both source and tag.
  - If source is set to MPI_ANY_SOURCE, then any process of the communication domain can be the source of the message.
  - Similarly, if tag is set to MPI_ANY_TAG, then messages with any tag are accepted.
- The received message is stored in continuous locations in the buffer pointed to by *buf*. The *count* and *datatype* arguments of MPI_Recv are used to specify the length of the supplied buffer. The received message should be of length equal to or less than this length.
- This allows the receiving process to not know the exact size of the message being sent. If the received message is larger than the supplied buffer, then an overflow error will occur, and the routine will return the error MPI_ERR_TRUNCATE.
- After a message has been received, the status variable can be used to get information about the MPI_Recv operation. In C, status is stored using the MPI_Status data-structure. This is implemented as a structure with three fields, as follows:

  ```
  typedef struct MPI_Status {
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
  };
  ```
- MPI_SOURCE and MPI_TAG store the source and the tag of the received message. They are particularly useful when MPI_ANY_SOURCE and MPI_ANY_TAG are used for the source and tag arguments. MPI_ERROR stores the error-code of the received message.
- The status argument also returns information about the length of the received message. This information is not directly accessible from the status variable, but it can be retrieved by calling the MPI_Get_count function. The calling sequence of this function is as follows:

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype,
        int *count)
```

MPI_Get_count takes as arguments the status returned by MPI_Recv and the type of the received data in datatype, and returns the number of entries that were actually received in the count variable.

- The MPI_Recv returns only after the requested message has been received and copied into the buffer. That is, MPI_Recv is a **blocking** receive operation.
- However, MPI allows two different implementations for MPI_Send.
    1. MPI_Send returns only after the corresponding MPI_Recv have been issued and the message has been sent to the receiver.
    2. MPI_Send first copies the message into a buffer and then returns, without waiting for the corresponding MPI_Recv to be executed.
- In either implementation, the buffer that is pointed by the buf argument of MPI_Send can be safely reused and overwritten.
- MPI programs must be able to run correctly regardless of which of the two methods is used for implementing MPI_Send. Such programs are called safe.
- In writing safe MPI programs, sometimes it is helpful to forget about the alternate implementation of MPI_Send and just think of it as being a blocking send operation.