# Message Passing Interface - MPI

## What is MPI?

Message passing is a programming paradigm used widely on parallel computer architectures and networks of workstations. One of the attractions of the paradigm is that it will not be made obsolete by architectures combining the shared- and distributed-memory views, or by increases in network speeds.

The MPI standard defines the user interface and functionality, in terms of syntax and semantics, of a standard core of library routines for a wide range of message-passing capabilities. It defines the logic of the system but is not implementation specific.

The specification can be efficiently implemented on a wide range of computer architectures. It can run on distributed-memory parallel computers, a shared-memory parallel computer a network of workstations, or, indeed, as a set of processes running on a single workstation. The standard has come from a unification of concepts and most attractive ideas from vendor specific message passing system variants through the work of the MPI forum.

### Goals

To design an application programming interface which can achieve portability across a number of parallel architectures and reflects the needs of application programmers. Same source code can work provided MPI library is available for the architecture. For example, an MPI implementation on top of standard Unix interprocessor communication protocols will provide portability to workstation clusters and heterogeneous networks of workstations.

The semantics of the interface should be language independent to allow for convenient bindings for common programming languages like C and FORTRAN used in high performance computing. MPI can also be used as a run-time for parallel compilers.

Add credibility to parallel computing. MPI has become widely accepted and used as it provides an interface similar to existing vendor practice but with additional features and extensions.

Provide a virtual computing model that hides many architectural differences and allows execution on heterogeneous systems. For example, an MPI implementation will automatically do any necessary data conversion and utilize the correct communications protocol.

MPI was carefully designed so as to make efficient implementations possible without significant changes in the underlying communication and system software. Portability is central but the standard would not gain wide usage if this were achieved at the expense of performance. Some communication efficiency principles include avoiding memory to memory copying, allowing asynchronous overlap with computation, and offloading to a communication processor where available.

Scalability is an important goal of parallel processing. MPI supports scalability through several of its design features. For example, an application can create subgroups of processes that, in turn, allow collective communication operations to limit their scope to the processes involved.

Define a reliable interface with known minimum behaviour for message-passing implementations. This relieves the programmer from having to cope with communication failures.

**Overview of The MPI Standard Functionality**

***Point to Point Communication***
MPI provides a set of send and receive functions that allow the communication of typed data with an associated message tag Typing of the message contents is necessary for heterogeneous support. The type information is needed so that correct data representation conversions can be performed as data is sent from one architecture to another. The tag allows selectivity of messages at the receiving end. One can receive on a particular tag, or one can wild-card this quantity, allowing reception of messages with any tag. Message selectivity on the source process of the message is also provided.

***Collective Operations***
Collective communications transmit data among all processes in a group specified by an intracommunicator object. One function, the barrier function, serves to synchronize processes without passing data. No process returns from the barrier function until all processes in the group have called it. A barrier is a simple way of separating two phases of computation to ensure that the messages generated in the two phases do not intermingle. MPI provides the following collective communication functions.

Barrier synchronization across all group members
Global communication functions – Data Movement Routines
- Broadcast of same data from one member to all members of a group
- Gather data from all group members to one member
- Scatter different data from one member to other members of a group
- A variation on Gather where all members of the group receive the result
- Scatter/Gather data from all members to all members of a group (also called complete exchange or all-to-all)

Global reduction operations such as sum and product, max and min, bitwise and logical, or user-defined functions.
-Reduction where the result is returned to all group members and a variation where the result is returned to one member
-A combined reduction and scatter operation

***Process Groups***
In some applications it is desirable to divide up the processes to allow different groups of processes to perform independent work. A group is an ordered set of process identifiers. Every process in a group is associated with an integer rank. Ranks are contiguous and start from zero. MPI provides functionality for constructing and destructing groups of processes and for accessing information about group membership.

***Communication Domains***
A **communicator** object specifies a communication domain which can be used for point-to-point communications.

An **intracommunicator** is used for communicating within a single group of processes. The intracommunicator has fixed attributes, for example, that describe the process

group and the topology of the processes in the group. Intracommunicators are also used for collective operations within a group of processes.

An **intercommunicator** is used for point-to-point communication between two disjoint groups of processes. The fixed attributes of an intercommunicator are the two groups. No topology is associated with an intercommunicator.

***Process Topologies***
A process group in MPI is a collection of `n` processes. Each process in the group is assigned a rank between `0` and `n-1`. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes within the group.

A topology can provide a convenient naming mechanism for the processes of a group and may assist the runtime system in mapping the processes onto hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors, if this helps to improve the communication performance on a given machine.

A large amount of parallel applications arrange processes in topological patterns such as two- or three-dimensional grids. More generally, the logical process arrangement, or virtual topology, can be described by a graph. MPI provides functionality for constructing both Cartesian form and Graph Topologies.

***Environmental Management and Enquiry***
One goal of MPI is to achieve source code portability. A program written using MPI and complying with the relevant language standards is portable as written, and must not require any source code changes when moved from one system to another. This does *not* say anything about how an MPI program is started or launched from the command line, or what the user must do to set up the environment in which an MPI program will run. However, an implementation may require some setup to be performed before other MPI routines may be called. To provide for this, MPI includes an initialisation routine MPI_INIT and termination routines. A set of attributes that describe the execution environment are attached to the communicator MPI_COMM_WORLD when MPI is initialized.

An MPI implementation may or may not choose to handle some errors that occur during MPI calls. Each such error generates an MPI exception. A good quality implementation will attempt to handle as many errors as possible. Errors that are not handled by MPI will be handled by the error handling mechanisms of the language run-time or the operating system. Typically, errors that are not handled by MPI will cause the parallel program to abort.

A program error can occur, for example, when an MPI routine is called with an incorrect argument. Message delivery errors may occur with the underlying communication system. A resource error can occur when an activity exceeds the amount of available system resources. The occurrence of this type of error depends on the amount of available resources in the system and the resource allocation mechanism used. A high-quality implementation will provide generous limits on the important resources so as to alleviate the portability problem this represents.

A user can associate an error handler with a communicator. The specified error handling routine will be used for any MPI exception that occurs during a call to MPI for a communication with this communicator.

***Profiling Interface***
Profiling is the gathering of a program's performance characteristics by measuring events happening in a computer system while running the program. The output is a stream or summary of the recorded events

Since MPI is a machine independent standard with many different implementations, it is unreasonable to expect that the authors of profiling tools for MPI will have access to the source code which implements MPI on any particular architecture. It is therefore necessary to provide a mechanism by which the implementers of such tools can collect whatever performance information they wish without access to the underlying implementation.

The user code must able to control the profiler dynamically at run time. This is useful for purposes such as:-

Enabling and disabling profiling depending on the state of the calculation
Flushing trace buffers at non-critical points in the calculation
Adding user events to a trace file.

These requirements are met by use of the MPI_PCONTROL function.

Such an interface may also prove useful for other purposes, such as ``internetworking" multiple MPI implementations.

**MPI is a Big Interface**
One aspect of concern, particularly to novices, is the large number of routines and parameters comprising the MPI specification. Firstly, MPI was designed to be rich in functionality. This is reflected in MPI's support for data types, modular communication, caching, application topologies, and the fully-featured set of collective communication routines. Secondly, this functionality is offered in a range of modes suited to the diversity and complexity of today's high performance computers. One could decrease the number of functions by increasing the number of parameters in each call. But such approach would increase the call overhead and would make the use of the most prevalent calls more complex. The availability of a large number of calls to deal with more esoteric features of MPI allows one to provide a simpler interface to the more frequently used functions.

**MPI Basics**
Although MPI is a complex and multifaceted system, we can solve a wide range of problems using just six of its functions! We introduce MPI by describing these six functions, which initiate and terminate a computation, identify processes, and send and receive messages:

For the function parameters, the labels `IN`, `OUT`, and `INOUT` indicate whether the function uses but does not modify the parameter (`IN`), does not use but may update the parameter (`OUT`), or both uses and updates the parameter (`INOUT`).

```
MPI_INIT(int *argc, char ***argv)
Initiate an MPI computation.
```

`argc and argv are required only in the C language binding where they are the main program's arguments.`

`MPI_FINALIZE`()
Terminate and shut down a computation.

`MPI_COMM_SIZE(comm, size)`
Determine number of processes in a computation.

IN comm communicator (handle)
OUT size number of processes in the group

`MPI_COMM_RANK`(comm, pid)
Determine the process identifier of the current process.

IN comm communicator (handle)
OUT pid process id in the group of comm

`MPI_SEND`(buf, count, datatype, dest, tag, comm)
Send a message. A message containing count elements of the specified data type starting at address buf is to be sent to the process with the identifier dest. The message is associated with an envelope comprising the specified tag, the source process's identifier and the specified communicator.

IN buf address of send buffer (choice)
IN count number of elements to send (integer >=0)
IN datatype data type of send buffer elements (handle)
IN dest process id of destination process (integer)
IN tag message tag (integer)
IN comm communicator (handle)

`MPI_RECV(buf, count, datatype, source, tag, comm, status)`
Receive a message. Attempts to receive a message that has an envelope corresponding to the specified tag, source and communicator, blocking until such a message is available. When the message arrives, elements of the specified datatype are placed into the buffer at address buf. Buffer is large enough to hold at least count elements. The status variable can be used to inquire about the size, tag and source of the received message.

OUT buf address of receiver buffer (choice)
IN count size of receive buffer, in elements (integer >=0)
IN datatype datatype of receive buffer elements (handle)
IN source process id of source process or MPI_ANY_SOURCE (integer)
IN tag message tag or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT status status object (status)

All but the first two calls take a communicator handle as an argument. Communicators provide a mechanism for identifying process subsets during development of modular programs. A communicator identifies the process group and context with respect to which the operation is to be performed. For now, it suffices to provide the default value `MPI_COMM_WORLD`, which identifies *all* processes involved in a computation.

The functions `MPI_INIT` and `MPI_FINALIZE` are used to initiate and shut down an MPI computation, respectively. `MPI_INIT` must be called before any other MPI function and must be called exactly once per process. No further MPI functions can be called after `MPI_FINALIZE`.

**Example Program Sequence Without Communication**

```
program main

 begin
      MPI_INIT()                                        Initiate computation

      MPI_COMM_SIZE(MPI_COMM_WORLD, count)       Find # of processes

      MPI_COMM_RANK(MPI_COMM_WORLD, myid)    Find my id

      print("I am", myid, "of", count)                      Print message

      MPI_FINALIZE()                                    Shut down
 end
```

The MPI standard does not specify how a parallel computation is started. However, a typical mechanism could be a command line argument indicating the number of processes that are to be created:

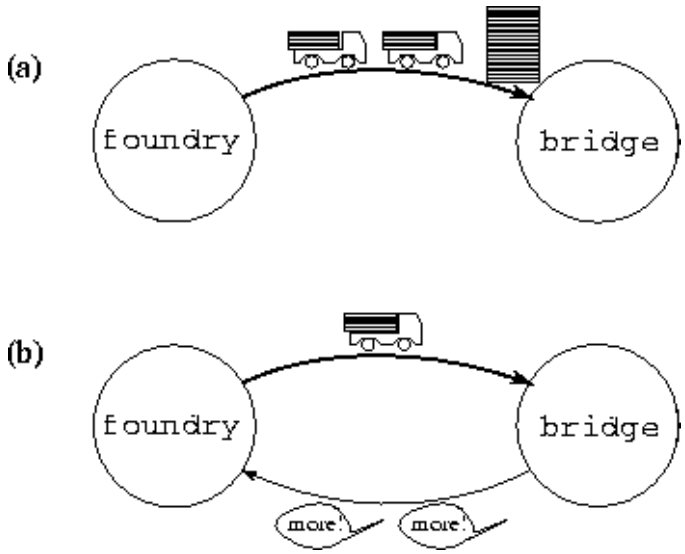
For example, `myprog -n 4`, where `myprog` is the name of the executable.

If the program above is executed by four processes, we will obtain something like the following output. The order in which the output appears is not defined; however, we assume here that the output from individual print statements is not interleaved.

```
I am 1 of 4
I am 3 of 4
I am 0 of 4
I am 2 of 4
```

**Parallel Programming Example (With Point to Point Communication)**
**Bridge Construction**

A bridge is to be assembled from girders being constructed at a foundry. These two activities are organized by providing trucks to transport girders from the foundry to the bridge site. This situation is illustrated in the figure overleaf with the foundry and bridge represented as tasks and the stream of trucks as a channel. Notice that this approach allows assembly of the bridge and construction of girders to proceed in parallel without any explicit coordination. The foundry crew puts girders on trucks as they are produced, and the assembly crew adds girders to the bridge as and when they arrive.




Two solutions to the bridge construction problem. Both represent the foundry and the bridge assembly site as separate tasks.

The first uses a single channel on which girders generated by `foundry` are transported as fast as they are generated. If `foundry` generates girders faster than they are consumed by `bridge`, then girders accumulate at the construction site.

The second solution uses a second channel to pass flow control messages from `bridge` to `foundry` so as to avoid overflow.

A coded implementation of the first solution using MPI is presented overleaf. A coded implementation for the second solution is left as an exercise.

**Code for Solution (a) – Requires buffering of messages from foundry**

```
program main
begin
        MPI_INIT()
        MPI_COMM_SIZE(MPI_COMM_WORLD, count)
        if count != 2 then exit                 /* Must be just 2 processes */
        MPI_COMM_RANK(MPI_COMM_WORLD, myid)
        if myid = 0 then
          foundry(100)                  /* Execute Foundry */
        else
          bridge()                              /* Execute Bridge */
        endif
        MPI_FINALISE()
end

procedure foundry(numgirders)
begin
        for i = 1 to numgirders
                /* Make a girder and send it */
                /* MPI_SEND(buf, count, datatype, dest, tag, comm) */
                MPI_SEND(i, 1, MPI_INT, 1, 0, MPI_COMM_WORLD)
        endfor
```

```
        i = -1              /* Send shutdown message */
        MPI_SEND(i, 1, MPI_INT, 1, 0, MPI_COMM_WORLD)
end

procedure bridge
begin
        /* Wait for girders and add them to the bridge */
        /* MPI_RECV(buf, count, datatype, source, tag, comm,
status) */
        MPI_RECV(msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, status)
        while msg != -1 do
                use_girder(msg)
                MPI_RECV(msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, status)
        endwhile
end
```

Processes can use point-to-point communications operations to send a message from one named process to another. These operations are used to implement **local** and unstructured communications.

Parallel algorithms often call for coordinated communication operations involving multiple processes. A group of processes can call collective communication operations to perform commonly used **global** operations such as summation and broadcast. These MPI operations are discussed next.

**Global Operations**

***Global Synchronisation***

MPI_BARRIER(comm)
IN comm communicator(handle)

This function is used to synchronise execution of a group of processes. No process returns from this function until all processes have called it. A barrier is a simple way of separating two phases of computation to ensure that the messages generated in the two phases do not intermingle. In many cases, the need for a barrier can be avoided with appropriate use of tags and source specifiers.

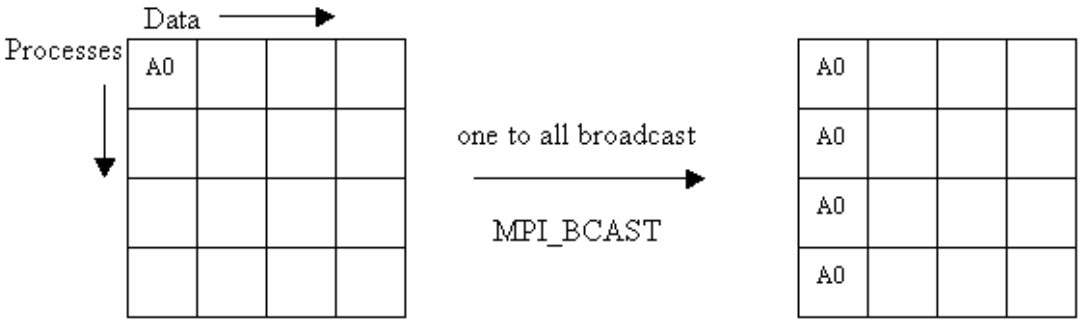
***Collective Data Movement***

MPI_BCAST, MPI_GATHER and MPI_SCATTER are collective data movement routines in which all processes interact with a distinguished root process. In each case, the first three parameters specify the location (inbuf) and type (intype) of the data to be communicated and the number of elements to be sent to each destination (incnt). Other parameters specify the location and type of the result (outbuf and outtype) and the number of elements to be received from each source (outcnt).

MPI_BCAST(inbuf, incnt, intype, root, comm)
INOUT inbuf address of input buffer, or output buffer at root (choice)
IN incnt number of elements in input buffer (integer)
IN intype datatype of input buffer elements (handle)
IN root process id of root process (integer)
IN comm communicator (handle)

This function implements a one-to-all broadcast where a single named root process sends the same data to all other processes. At the time of the call, the data is located in

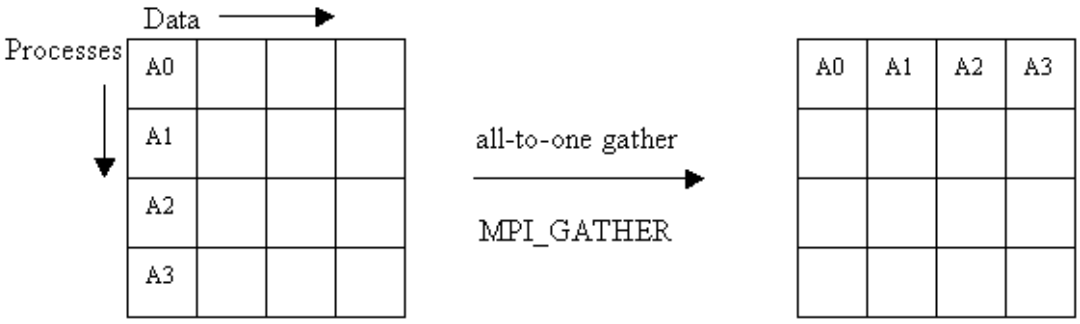
inbuf in process root and consists of incnt items of type intype. After the call the data is replicated in inbuf in all processes. As inbuf is used for input at the root and for output in other processes, it has type INOUT.




MPI_GATHER(inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)

| | | |
|---|---|---|
| IN | inbuf | address of input buffer (choice) |
| IN | incnt | number of elements sent to each (integer) |
| IN | intype | datatype of input buffer elements (handle) |
| OUT | outbuf | address of output buffer (choice) |
| IN | outcnt | number of elements received from each (integer) |
| IN | outtype | datatype of output buffer elements (handle) |
| IN | root | process id of root process (integer) |
| IN | comm | communicator (handle) |

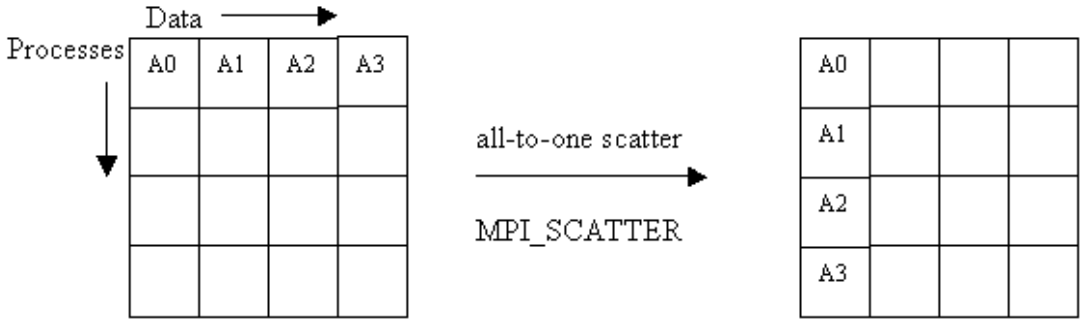
This function implements an all-to-one gather operation. All processes (including the root process) send data located in inbuf to root. The process places the data in contiguous non overlapping locations in outbuf, with the data from process i preceding that from process i+1. The outbuf in the root process must be P times larger than inbuf, where P is the number of processes participating. The outbuf in processes other than root is ignored.



MPI_SCATTER(inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)

| | | |
|---|---|---|
| IN | inbuf | address of input buffer (choice) |
| IN | incnt | number of elements sent to each (integer) |
| IN | intype | datatype of input buffer elements (handle) |
| OUT | outbuf | address of output buffer (choice) |
| IN | outcnt | number of elements received from each (integer) |
| IN | outtype | datatype of output buffer elements (handle) |
| IN | root | process id of root process (integer) |
| IN | comm | communicator (handle) |

The scatter operation is the reverse of MPI_GATHER. A specified root process sends data to all processes, sending the ith portion of its inbuf to process i; each process receives data from root in outbuf. Hence the inbuf in the root process must be P times larger than outbuf. This function differs from MPI_BCAST in that every process receives a different value.



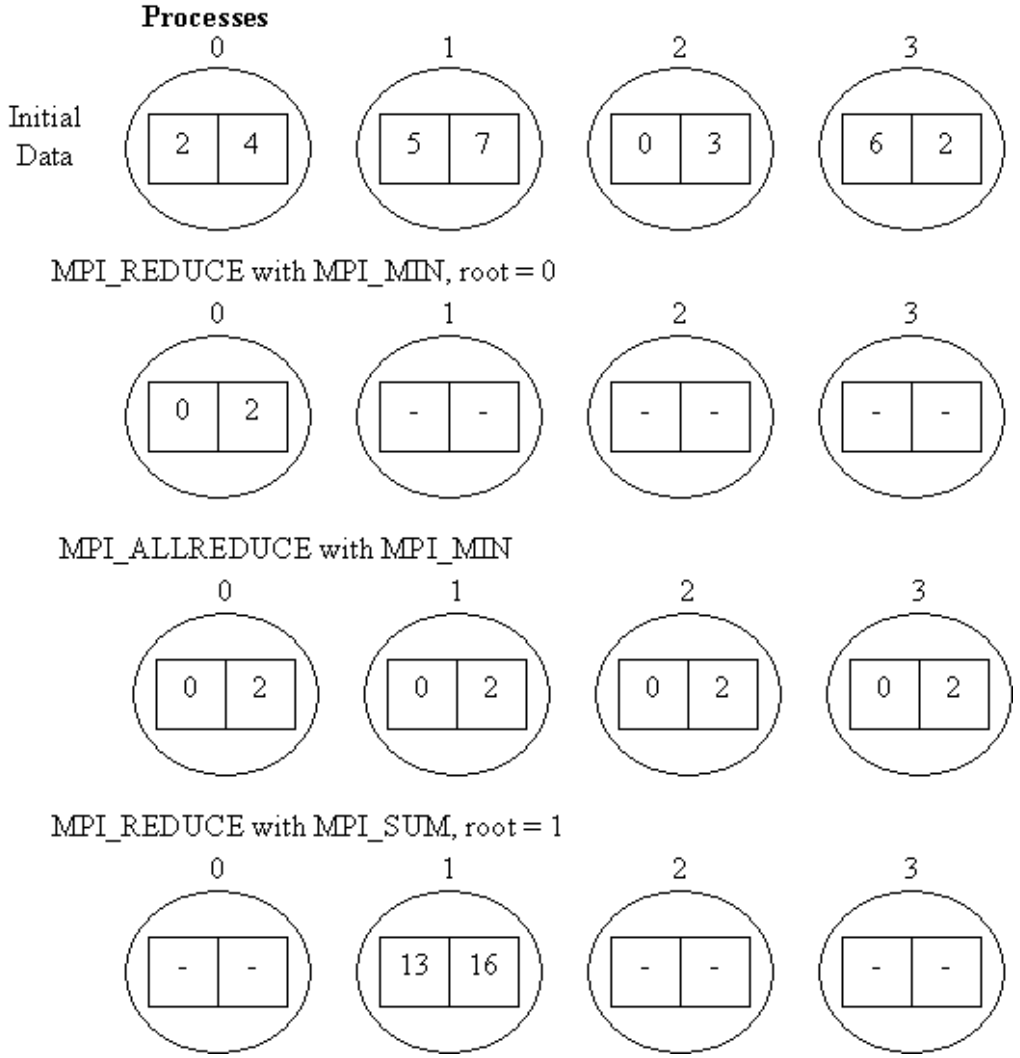

***Reduction Operations***

The functions MPI_REDUCE and MPI_ALLREDUCE implement reduction operations. They combine the values provided in the input buffer of each process, using a specified operation op, and return the combined value either to the output buffer of the single root process (in the case of MPI_REDUCE) or to the output buffer of all processes (MPI_ALLREDUCE). The operation is applied pointwise to each of the count values provided by each process. All operations return count values with the same datatype as the operands.

MPI_REDUCE(inbuf, outbuf, count, type, op, root, comm)
MPI_ALLREDUCE(inbuf, outbuf, count, type, op, comm)

| | | |
|---|---|---|
| IN | inbuf | address of input buffer (choice) |
| OUT | outbuf | address of output buffer (choice) |
| IN | count | number of elements in input buffer (integer) |
| IN | type | datatype of input buffer elements (handle) |
| IN | op | reduction operation (handle) |
| IN | root | process id of root process (integer) |
| IN | comm | communicator (handle) |

Valid operations include maximum and minimum (MPI_MAX and MPI_MIN); sum and product (MPI_SUM and MPI_PROD); logical and, or and exclusive or (MPI_LAND, MPI_LOR and MPI_LXOR; bitwise and, or and exclusive or (MPI_BAND, MPI_BOR and MPI_BXOR).




**Finite Differences**

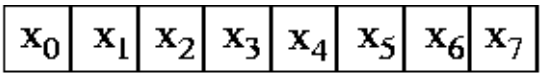
A parallel algorithm for the one-dimensional finite difference problem.

We consider a one-dimensional finite difference problem, in which we have a vector $X^{(0)}$ of size N and, after T iterations, compute a vector $X^{(T)}$, where a difference function is applied to the $i^{th}$ element of X, on each iteration t. The algorithm terminates

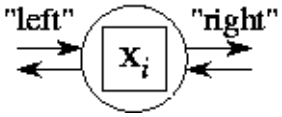
when the difference between any two neighbours reaches an acceptable minimum value.

We must repeatedly update each element of X, with no element being updated in step t+1 until its neighbours have been updated in step t.

A parallel algorithm for this problem creates N tasks, one for each point in X.
In the diagram below, from top to bottom, we have the one-dimensional vector X, where N=8; the task structure, showing the 8 tasks, each encapsulating a single data value and connected to left and right neighbours via channels; and the structure of a single task, showing its two in-ports and out-ports.



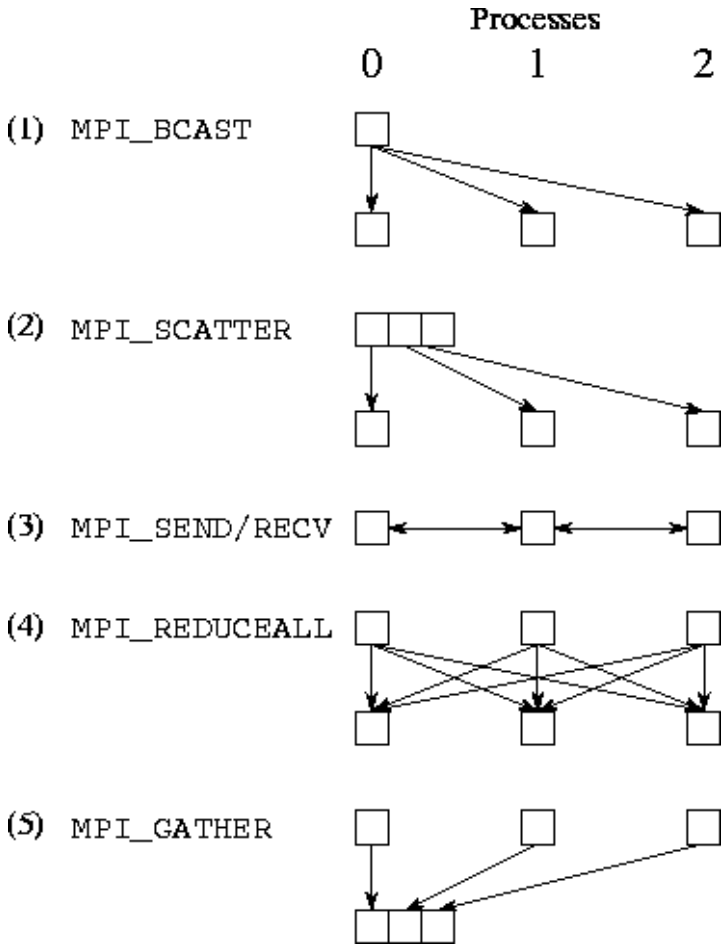





The $i^{th}$ task is given the value $X_i^{(0)}$ and is responsible for computing the values $X_i^{(1)}$, $X_i^{(2)}$, ..., $X_i^{(T)}$ and so on. Hence, at step t, it must obtain the values $X_{i-1}^{(t)}$ and $X_{i+1}^{(t)}$ from tasks i-1 and i+1. We specify this data transfer by defining channels that link each task with ‘left’ and ‘right’ neighbours, as shown above, and requiring that at step t, each task i do the following:-

1. sends its data $X_i^{(t)}$ on its left and right out-ports,
2. receives $X_{i-1}^{(t)}$ and $X_{i+1}^{(t)}$ from its left and right in-ports, and
3. uses these values to compute $X_i^{(t+1)}$.

Notice that the N tasks can execute independently, with the only constraint on execution order being the synchronization enforced by the receive operations. This synchronization ensures that no data value is updated at step t+1 until the data values in neighbouring tasks have been updated at step t.
The algorithm requires both nearest-neighbour communication (to exchange boundary values) and global communication (to initialise the computation and detect termination). It uses MPI_SEND and MPI_RECV for nearest-neighbour communication and four MPI global communication routines, for a total of five distinct communication operations. These are summarized as follows:




Communication performed in the finite difference program, assuming three processes. Each column represents a processor; each subfigure shows data movement in a single phase.

The five phases illustrated are (1) broadcast, (2) scatter, (3) nearest-neighbour exchange, (4) reduction, and (5) gather.

MPI_BCAST to broadcast the problem size parameter (size) from process 0 to all np processes;

MPI_SCATTER to distribute an input array (work) from process 0 to other processes, so that each process receives size/np elements;

MPI_SEND and MPI_RECV for exchange of data (a single floating-point number) with neighbours;

MPI_ALLREDUCE to determine the maximum of a set of localerr values computed at the different processes and to distribute this maximum value to each process; and

MPI_GATHER to accumulate an output array at process 0.

```
main(int argc, char *argv[]) {
```

```
   MPI_Comm com = MPI_COMM_WORLD;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(com, &np);  /* number of processes in computation */
   MPI_Comm_rank(com, &me);

   if (me == 0) {       /* Read problem size at process 0 */
      read_problem_size(&size);
      buff[0] = size;
   }

   /* Phase 1 - Broadcast problem size to all processes. */
   MPI_Bcast(buff, 1, MPI_INT, 0, com);

   /* Extract problem size from buff */
   lsize = buff[0]/np;

   /* Allocate space for local data */
   local = malloc(lsize);

   /* Read input data at process 0 */
   if (me == 0) {
      work = malloc(size);
      read_array(work);
   }
   /* Phase 2 - Distribute the data elements to the processes. */
   MPI_Scatter(work,lsize,MPI_FLOAT,local,lsize,MPI_FLOAT,0,com);



 /* Determine my neighbours in ring */
   lnbr = (me+np-1)%np;
   rnbr = (me+1) %np;

   globalerr = 99999.0;
   while (globalerr > 0.1) /* Repeat until termination */
      /* Phase 3 - Exchange boundary values with neighbours */

      MPI_Send(local, 1, MPI_FLOAT, lnbr, 10, com);
      MPI_Send(local, 1, MPI_FLOAT, rnbr, 20, com);
      MPI_Recv(left, 1,MPI_FLOAT, lnbr, 20, com, &status);
      MPI_Recv(right, 1, MPI_FLOAT, rnbr, 10, com, &status);
      Compute(local, left, right);
      localerr = maxerror(local, left, right);

      /* Phase 4 - Find maximum local error and replicate in each
         process */
      MPI_Allreduce(&localerr,&globalerr,1,MPI_FLOAT,MPI_MAX,com);
   }
   /* Phase 5 - Collect results at process 0 */
   MPI_Gather(local,lsize, MPI_FLOAT, work, size, MPI_FLOAT, 0, com);
   if (me == 0) {
      write_array(work);
      free(work);
   }
   MPI_Finalize();
}
```

The use of scatter and gather operations to transfer input and output data is particularly simple and convenient. Note, however, that their use in this example is inherently nonscalable. As we solve larger problems, storage limitations will eventually prevent us from accumulating all input and output data in a single process. In addition, the associated communication costs may be prohibitive.