1. Chapter 3 deals with distributed-memory programming with MPI (*Message Passing Interface*). MPI is a distributed memory programming model in which a collection of processes communicate by sending messages.

```
/*  File: sum1DArray.c
 *  Compile as:  mpicc -o sum1DArray sum1DArray.c
 *  Run as:  mpirun -np 4 -hostfile nodes ./sum1DArray 1024
 *  Description:  An MPI solution to sum a 1D array. */

#include <stdlib.h>
#define RootProcess 0

#include <sys/types.h>
#include <time.h>
#include <stdio.h>
#include <mpi.h>

const int tag = 1;

int main(int argc, char* argv[]) {
  int myID, value, numProcs, i, p;
  float * myArray;
  double seqSum, parallelSum, localSum;
  int length;
  int length_per_process;
  clock_t clockStart, clockEnd;

  clockStart = clock();

  MPI_Status status;

  MPI_Init(&argc, &argv);  /* Initialize MPI */
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs); /* Get rank */
  MPI_Comm_rank(MPI_COMM_WORLD, &myID); /* Get rank */

  // all processes have access to argc and argv
  sscanf(argv[1],"%d",&length);
  length_per_process = length/numProcs;

  if ( myID == RootProcess ) {       /* Broadcast length */
    if (argc != 2) {
      printf("Usage: %s <# of floats to sum>\n", argv[0]);
      return(0);
    };

    // Generate data array   length = SIZE;
    printf("length = %d\n", length);
    myArray=(float *) malloc(length*sizeof(float));
    srand(5);
    for (i=0; i < length; i++) {
      myArray[i] = rand() / (float) RAND_MAX;
    } // end for i

    /* Send a message with part of array to each MPI process*/
    for (p=0; p<numProcs-1; p++) {
      MPI_Send( myArray+length_per_process*p, length_per_process, MPI_FLOAT, p+1, tag, MPI_COMM_WORLD );
    } // end for p

    /* Do the actual work - do "right" end of array*/
    localSum = 0.0;
    for (i= length_per_process*(numProcs-1);i < length; i++) {
      localSum += myArray[i];
    } // end for i
```

```
  } else {  // code for MPI processes that are not the root process
    myArray = (float *) malloc(length_per_process*sizeof(float));
    MPI_Recv( myArray, length_per_process, MPI_FLOAT, RootProcess, tag, MPI_COMM_WORLD, &status );

    /* Do the actual work */
    localSum = 0.0;
    for (i=0; i < length_per_process; i++) {
      localSum += myArray[i];
    } // end for i

  } // end if
  MPI_Reduce(&localSum, &parallelSum, 1, MPI_DOUBLE, MPI_SUM, RootProcess, MPI_COMM_WORLD);
  clockEnd = clock();

  if (myID == RootProcess) {
     printf( "Time to sum %d floats with MPI in parallel  %3.5f seconds\n", length,
              (clockEnd - clockStart) / (float) CLOCKS_PER_SEC);
    clockStart = clock();
    seqSum = 0.0;
    for (i=0; i < length; i++) {
      seqSum += myArray[i];
    } // end for i
    clockEnd = clock();
    printf( "Time to sum %d floats sequentially  %3.5f seconds\n", length,
            (clockEnd - clockStart) / (float) CLOCKS_PER_SEC);

    printf("The parallel sum:   %f\n", parallelSum);
    printf("The sequential sum: %f\n", seqSum);

  } // end if
  free(myArray);

  MPI_Finalize();
  return 0;
} /* end main */
```

MPI_Init()

```
int MPI_Init(                          // Initialize MPI
  int *argc,                           // Number of command line arguments
  char ***argv,                        // Command line arguments
);
```

Arguments:

- Number of command line arguments.
- Command line arguments.

Notes:

This routine must be called in every MPI process before any other MPI routine is called. It is an error to call this routine more than once in a process unless a subsequent `MPI_Finalize()` is called.

Return value:

An MPI error code.

MPI_Finalize()

```
int MPI_Finalize(
);
```

Notes:

This routine should be the last MPI routine called in each process, and it should only be invoked after all other MPI routines have completed. In particular, any pending communication operations should complete before this routine is called.

Return value:

An MPI error code.

MPI_Comm_Size()

```
int MPI_Comm_Size(                     // Retrieve the number of tasks in
  MPI_Comm comm,                       // the specified communicator
  int *size,                           // The number of tasks
);
```

Arguments:

- The communicator of interest.
- A pointer to the size, whose target will contain the number of tasks in the specified communicator.

Notes:

This routine obtains the number of processes in a communicator.

Return value:

An MPI error code.

MPI_Comm_Rank()

```
int MPI_Comm_Rank(                     // Retrieve rank of a communicator
  MPI_Comm comm,                       // Communicator
  int *rank,                           // Rank
);
```

Notes:

This routine obtains a process' rank within a communicator.

Arguments:

- The communicator of interest.
- A pointer to the rank, whose target will contain the rank of the specified communicator.

Return value:

An MPI error code

MPI_Send()

```
int MPI_Send(                          // Blocking Send routine
  void         *buffer,                // Address of the data to send
  int          count,                  // Number of data elements to send
  MPI_Datatype type,                   // Type of data elements to send
  int          dest,                   // ID of destination process
  int          tag,                    // Tag to distinguish this message
  MPI_Comm     *comm                   // An MPI communicator
);
```

Arguments:

- The address of the data to send.
- The number of data elements to send.
- The type of data elements to send.
- The ID of the process that should receive this message.
- A message tag that distinguishes this message from others that may be sent to the same process.
- The MPI communicator to use.

Notes:

This routine sends data to another process. This routine has blocking semantics, which means that the routine does not return until the message has been sent. `MPI_Isend()` is a non-blocking version of the send operation; it takes a seventh parameter of type `MPI_Request` that is used to differentiate this send from other invocations of `MPI_Isend()`when waiting for completion.

Return value:

An MPI error code.

MPI_Recv()

```
int MPI_Recv(                          // Blocking Receive routine
  void         *buffer,                // Address at which to receive data
  int          count,                  // Number of elements to receive
  MPI_Datatype type,                   // Type of each element
  int          source,                 // ID of sending process
  int          tag,                    // Identifier to distinguish message
  MPI_Comm     comm,                   // MPI communicator
  MPI_Status   *status                 // Status of this receive operation
);
```

Arguments:

- The first six arguments correspond to `MPI_Send()`
- To receive a message from any other process, use `MPI_ANY_SOURCE` as the `source`.
- To match on any tag, use `MPI_ANY_TAG` as the fifth parameter.

Notes:

This routine receives data from another process. This routine has blocking semantics—it does not return until the message is received. `MPI_Irecv()` is a non-blocking version of the receive operation; it takes a seventh parameter of type `MPI_Request` that is used to differentiate this receive from other invocations of `MPI_Irecv()`when waiting for completion.

Return value:

An MPI error code.

MPI_Reduce()

```
int MPI_Reduce(                        // Reduce routine
  void         *sendBuffer,            // Address at which to receive data
  void         *recvBuffer,            // Number of elements to receive
  int          count,                  // Type of each element
  MPI_Datatype datatype,               // ID of sending process
  MPI_OP       op,                     // MPI operator
  int          root,                   // Process that will contain result
  MPI_Comm     comm                    // MPI communicator
);
```

Notes:

This routine implements a reduce operation. A special form of this routine, `MPI_Allreduce()`, treats all processes as if they were the root, meaning that the reduced value will be passed to all processes at the address specified by the second argument. `MPI_Allreduce()` is equivalent to a call to `MPI_Reduce()` followed by a call to `MPI_Bcast()`, which broadcasts values to all processes within a communicator.

Return value:

An MPI error code.

MPI_Bcast()

```
int MPI_Bcast(                         // Broadcast routine
  void         *buffer,                // Address of the data to send
  int          count,                  // Number of data elements to send
  MPI_Datatype datatype,               // Type of data elements to send
  int          root,                   // Rank of the root task
  MPI_Comm     *comm                   // An MPI communicator
);
```

Arguments:

- The first three arguments specify the address, size, and type of the data elements to send to each process.
- The fourth argument specifies the rank of the root, or sending, process.
- The fifth argument specifies MPI communicator to use.

Notes:

This routine broadcasts data from the root process to all other processes in the communicator. Unlike `MPI_Scatter()` and `MPI_Gather()`, the number of elements and the types of the elements must be the same for the root process and the receiving processes.

Return value:

An MPI error code.

MPI_Scatter could be used to send equal size blocks to each process

MPI_Scatter()

```
int MPI_Scatter(                        // Scatter routine
  void          *sendbuffer,            // Address of the data to send
  int           sendcount,              // Number of data elements to send
  MPI_Datatype  sendtype,               // Type of data elements to send
  int           destbuffer,             // Address of buffer to receive data
  int           destcount,              // Number of data elements to receive
  MPI_Datatype  desttype,               // Type of data elements to receive
  int           root,                   // Rank of the root process
  MPI_Comm      *comm                   // An MPI communicator
);
```

Arguments:

- The first three arguments specify the address, size, and type of the data elements to send to each process. These arguments only have meaning for the root process.
- The second three arguments specify the address, size, and type of the data elements for each receiving process. The size and type of the sending data and the receiving data may differ as a means of converting data types.
- The seventh argument specifies the root process that is the source of the data.
- The eighth argument specifies the MPI communicator to use.

Notes:

This routine distributes data from the root process to all other processes, including the root. A more sophisticated version of the routine, `MPI_Scatterv()`, allows the root process to send different amounts of data to the various processes. Details can be found in the MPI standard.

Return value:

An MPI error code.

MPI_Scatterv and MPI_Gatherv should be used if each process does not receive the same number of data items.

```
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype, void*
              recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

IN sendbuf address of send buffer (choice, significant only at root)
IN sendcounts non-negative integer array (of length group size) specifying the number of elements to send to each processor
IN displs integer array (of length group size). Entry i specifies the displacement (relative to sendbuf from which to take the outgoing data to process i
IN sendtype data type of send buffer elements (handle)
OUT recvbuf address of receive buffer (choice)
IN recvcount number of elements in receive buffer (non-negative integer)
IN recvtype data type of receive buffer elements (handle)
IN root rank of sending process (integer)
IN comm communicator (handle)