"*What we know is not much. What we do not know is immense.*"

Pierre Simon De Laplace (1749-1827)

Chapter 11

Miscellaneous Numerical Methods

The following sections contain a very brief introduction into selected miscellaneous topics and examples from some specialty areas of numerical methods and analysis. The topics selected reflect upon current research of the time. We give a very brief look at parallel computing and the message passing interface language MPI. We take a passing glance at such areas as integral equations, Bézier curves, and B-splines. These are some areas of numerical methods and analysis that you might want to study in more depth. The material presented is far from complete and should be viewed only as introductory examples which are intended to serve as motivation for some readers to get more involved in the subject areas.

Parallel Computer Systems

Imagine that you have a computer code to run and instead of a single computer executing your computer program you had N $(N > 1)$ computers hooked up in parallel so that they all could communicate with each other and exchange information. How would you make use of all this extra computing power? This concept of a cluster of computers all doing calculations and communicating with each other is called parallel computing. The concept of serial and parallel computing is illustrated in the figure 10-1. We will be interested in situations where computer codes can be broken up so that the computations can be divided among the many computers available. The information calculated by the cluster of parallel computers can then be shared to compute the desired output.

Assuming that all the computers in serial and parallel modes are the same, then what kind of reduction in computational speed can one expect when using parallel computing? It turns out that most computer codes run faster in a parallel mode as compared to a serial mode. The reduction in computing time depends upon the type and number of parallel computers, the kind of code being run and how much message passing is done between computers within the cluster. For example, if a computer code, which runs in a serial mode, is modified to run in a parallel mode, then you can define the ratio

$$\frac{\text{Parallel Computing Time}}{\text{Serial Computing Time}} = f,$$

where all computers are the same. In comparing the run times of the computer codes, one will probably obtain an approximation like $f = .3 + \beta/N < 1$ for some positive constant β. Don't think that because you have N computers in parallel, that f will be around $1/N$ as this is unrealistic. It is the message passing that slows things down.

Not all computer codes are amenable for conversion to run under a parallel structure. Sometimes it is wiser to spend your money on a good serial computer. The kind of computer structure "best" for you will depend heavily upon the type of applications you are using your computer system to solve.

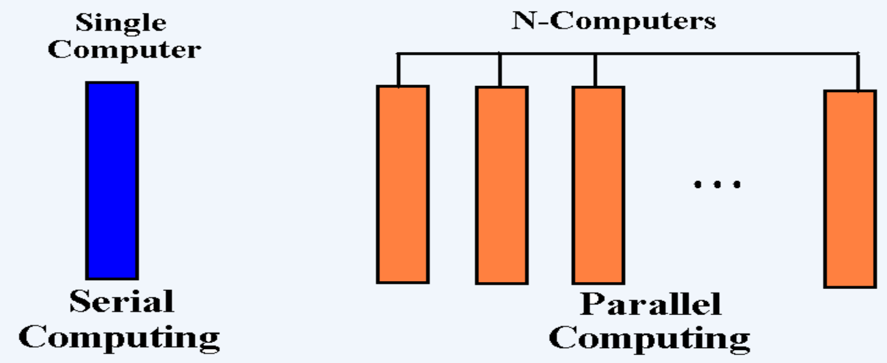

Figure 10-1 Serial and parallel computing

Building a parallel computer system is not hard. There are many individuals who collect old throw away computers and hook them up in a parallel structure. Loosely speaking any collection of computers that is capable of running a parallel code is called a Beowulf system or Beowulf cluster. (There are many definitions of a Beowulf system.) Hardware for connecting the computers comes in a variety of forms from inexpensive, with slow communication between computers, to expensive, with high speed communication between computers. The good news is that most software for multi-computer structures is free. For example, the MPI (Message-Passing Interface) libraries consist of Fortran, C and C++ language macros, functions and subroutines that can be inserted into Fortran, C and C++ computer codes to convert them to run within a parallel structure. There are many predefined constants built into the MPI libraries and one should note that variable names, function names and subroutine names beginning with MPI_ are protected and you should not change them. The following is a brief introduction into MPI programming for the Fortran computer language as this seems to be the language of choice of most individuals involved in scientific computing. As you begin your study of parallel computing you might think about how parallel computer systems can be employed to speed up numerical procedures. You will find by examining the scientific literature that parallel computing is being applied to almost all areas of numerical computing and new ideas are welcomed.

I cannot tell you how to compile and execute your MPI parallel code. At this time the commands for compiling and executing a parallel computer code varies widely from one computer system to another. Compiling and running codes for a parallel cluster also depends upon how environment variables are set up for your computer system. I suggest you contact the administrator of the parallel system you will be using to find out the compile and execute commands associated with your parallel computer system. On some unix systems the compile and run commands have the form

```
f77 -o compiledprogram fortranprogram.f -lmpi
mpirun -np 6 compiledprogram
```

where $-np$ is a mpirun processor flag which is followed by an integer representing the number of processor required to run the compiled code. The number 6 was selected as an example.

The following Fortran programs contain a small selection of MPI commands with a brief explanation as to their function. I will not go into detail about the function of each MPI command, but will leave it to the reader to purchase a text on MPI programming to find out the details associated with all the message passing between computers. The following computer codes are given strictly as an introduction for the purpose of illustrating selected applications and usages of parallel computing. The illustrative examples are just to get you started into some of the introductory aspects of parallel computing by way of examples. I will leave the complexities associated with parallel computing for additional study.

Example 11-1. (Getting started.)
The first example program is a variation of the 'HELLO WORLD' program found in

many introductory MPI manuals. In this first example, and all of the subsequent examples, the MPI library commands are written in bold face type so that you take notice of them. In the first example program it is assumed that it is compiled to run on 6-processors. That is, a copy of the program is placed upon each of the processors. What each processor does is controlled by the program.

The first MPI command to recognize is the

INCLUDE 'mpif.h'

command which is the first directive to be placed at the beginning of the Fortran code. The file mpif.h contains all the necessary information for compiling an MPI Fortran code. At some place after this first directive and before any MPI commands are issued, there must occur the initialization statement

CALL MPI_INIT(IERR)

where IERR is an error code with 0 denoting that there is no error. At the end of the program, after all MPI library usage has ended, there must occur the finalization statement

CALL MPI_FINIALIZE(IERR).

This statement will clear up any MPI commands that have not been completed during the execution of the program.

The commands

CALL MPI_COMM_RANK(MPI_COMM_WORLD, ID-PROC, IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NP, IERR)

determines the rank of each processor (ID-PROC) and number of processors (NP). The parameter MPI_COMM_WORLD is a predefined integer variable which is associated with information about all the processors being used in the program execution. It is referred to as a communicator argument. The parameter ID-PROC is the identification number assigned to each processor at start up. For the first example program, we use 6 processors and so each processors is assigned its own identification number of 0,1,2,3,4,5. Sometimes the processor 0 is referred to as the master processor. The command MPI_COMM_SIZE returns the integer variable NP denoting the number of processors being used.

Always remember that a copy of the code goes to each processor and that each processor has its own identification number. The first example code tests to see if the correct number of processors are being used. It then has each processor write out its identification number. Note that the order of the output depends upon which processor finishes first. The bottom half of the program has the message 'GOOD NIGHT WORLD' sent from the processor with identification rank 0 to the other processors and illustrates the use of the commands

```
    CALL MPI_SEND(SENDDATA, ICOUNT, DATATYPE, IDESTIN, ITAG, ICOMM, IERR)
    CALL MPI_RECV(RECVDATA,ICOUNT, DATATYPE, ISOURCE, ITAG, ICOMM, STATUS, IERR)
```

used to send and receive information. The parameters SENDDATA, RECVDATA represent the data being sent or received. The parameters ICOUNT and DATATYPE are used for system identification of the data sent and received. The MPI data types are

```
MPI_INTEGER                      MPI_COMPLEX
MPI_REAL                         MPI_LOGICAL
MPI_DOUBLE_PRECISION             MPI_CHARACTER
```

The parameters IDESTIN and ISOURCE are integers specifying the ranks of the receiving and sending processors. The ITAG parameter and MPI_COMM_WORLD parameters represent an integer tag and communicator. The parameter STATUS has dimension 2 and contains information regarding the data actually received. The first element of STATUS represents the source and the second element represents the tag.

After each processor writes out its processor number and value of NP, the code has an IF statement which tests the processor identification number. If this number is zero, then processor 0 creates the message 'GOOD NIGHT WORLD. A do-loop is then created to send this message to the processors 1,2,3,4,5. Note that each processor is executing the program. When a processor encounters the IF statement testing the processor number, then if the processor identification number is not zero, the processor is told that it should set up to receive the message being sent from processor zero. Always remember, a copy of the Fortran code is sent to each processor and each processor is executing the same code. The code then tells each processor what it should do. In the first half of the example program each processor was told to write its identification number and the value of NP (total number of processors). The second half of the program tells processor zero to create a message and then send the message to the other processors. All the processors with rank greater than zero are told to get ready to receive a message. The parameters within the MPI_SEND and MPI_RECV commands are used to distinguish between data being sent and received. This is the message-passing part of MPI that slows down the over-all run time of the code. After receiving the message that was sent, each processor is told to write out its processor identification number, the value of NP and the message that was received. The MPI_FINALIZE(IERR) is a must statement at the end of the MPI usage.

The MPI library has around 125 commands built in. The type of commands and number of commands are changing year to year. The six basic commands

```
MPI_INIT                 MPI_SEND
MPI_COMM_RANK            MPI_RECV
MPI_COMM_SIZE            MPI_FINALIZE
```

are all that is needed for many of the elementary uses of parallel processors running MPI Fortran codes.

```
      PROGRAM EXAMPLE1
C        Assume 6 processors are being used
      CHARACTER *16 MESSAGE
      INCLUDE 'mpif.h'
      INTEGER STATUS
C
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, ID-PROC, IERR)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NP, IERR)
      IF(NP .NE. 6) STOP 'WRONG NUMBER OF PROCESSORS'
      WRITE(*,100) ID-PROC,NP
100   FORMAT(1X,' hello world! I am processor number ',I3,
     1   ' OUT OF A TOTAL OF ',I3, ' PROCESSORS')
      TAG=5
      IF(ID-PROC .EQ. 0) THEN
      MESSAGE='GOOD NIGHT WORLD'
      IDNO=NP-1
      DO 10 I=1,IDNO
      CALL MPI_SEND(MESSAGE, 16, MPI_CHARACTER, I, TAG,
     1    MPI_COMM_WORLD, IERR)
10    CONTINUE
      ELSE
      CALL MPI_RECV(MESSAGE, 16, MPI_CHARACTER, 0, TAG),
     1    MPI_COMM_WORLD, STATUS, IERR)
      ENDIF
      WRITE(*,101) ID-PROC,NP,MESSAGE
101   FORMAT(1X,'I am finished with processor number ',I3,
     1    ' OUT OF A TOTAL OF ',I3, ' PROCESSORS',1x,A16)
C
      CALL MPI_FINALIZE(IERR)
      STOP
      END
```

■

Example 11-2. (Another example.)

Another example of parallel programing, using the above commands, is given in the computer program example2. The example2 code assumes the use of 4 processors where each processor is told to run a different computer code.

```
      PROGRAM EXAMPLE2
C        Assume 4 processors are being used
      INCLUDE 'mpif.h'
C
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, ID, IERR)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NP, IERR)
C        ID is the processor number
C        NP is the number of processors
      IF(NP .NE. 4) STOP 'WRONG NUMBER OF PROCESSORS'
C        ID=0,1,2,3 are the processor numbers
C        job 1 done on processor 0, job 2 done on processor 1, etc.
      IF(ID .EQ. 0) THEN
      {put job number 1 here}
      ELSE IF (ID .EQ. 1) THEN
      {put job number 2 here}
      ELSE IF (ID .EQ. 2) THEN
      {put job number 3 here}
      ELSE
      {put job number 4 here}
      END IF
      CALL MPI_FINALIZE(IERR)
      STOP 'FINISHED'
      END
```

■

Example 11-3.

The next example evaluates the integral $I = \int_a^b f(x)\,dx$ using 32 processors. Therefore, we divide the integral I into 32 parts and write

$$I = \int_{a_0}^{b_0} f\,dx + \int_{a_1}^{b_1} f\,dx + \cdots + \int_{a_i}^{b_i} f\,dx + \cdots + \int_{a_{31}}^{b_{31}} f\,dx$$

where $a_0 = a$ and $b_{31} = b$. The ith integration $I_i = \int_{a_i}^{b_i} f(x)\,dx$ is to be performed on processor number i, $i = 0, 1, 2, \ldots, 31$. Here the distance $b_i - a_i$ is the same for each integral I_i and is given by

$$b_i - a_i = \frac{b-a}{32}.$$

Therefore, each processor can use its own identification number to calculate the local limits of integration a_i and b_i. The local starting value a_i is found from the relation

$$a_i = a + ID(b-a)/32$$

where ID is the processor identification number. The local upper limit b_i is found from the relation

$$b_i = a_i + (b-a)/32$$

for $i = 1, 2, \ldots, 31$. We can define a local step size $h = (b_i - a_i)/N$ for each processor and calculate the local integral I_i using the trapezoidal rule. For purposes of illustration we select to use $N = 30$ panels for each local integral. After each processor calculates the local integral the value obtained is sent back to processor zero where all the information is summed.

In the program example3, we have selected $a = 10$ and $b = 100$ for the lower and upper limit, these numbers can be changed. The subroutine for calculating the local area using the trapezoidal rule and the function being integrated needs to be defined and append to the program example3. The following subroutine and function statement are representative of these required items.

```
      SUBROUTINE TRAP(A, B, N, H, AREA)
C       Trapezoidal rule for area under curve FUN(x)
      EXTERNAL FUN
      Area=(FUN(A)+FUN(B))/2.
      x=A
      DO 100 I=1,N-1
      x=x+H
      AREA=AREA+FUN(X)
100     CONTINUE
      AREA=AREA*H
      RETURN
      END

      FUNCTION FUN(X)
C       Function to be integrated
      FUN={put your function here}
      RETURN
      END
      PROGRAM EXAMPLE3
C        Assume 32 processors are being used
      INTEGER STATUS
      PARAMETER( a=10 , b= 100 )
      PARAMETER(N=30, ITAG=10, IDESTIN=0)
      INCLUDE 'mpif.h'
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, ID, IERR)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NP, IERR)
C        ID=0, 1, 2, 3, . . . , 31 are the processor numbers
C        Local limits of integration are given by
      ai=a+ID*(b-a)/NP
      bi=ai+(b-a)/NP
C        Local step size H is
      H=(bi-ai)/N
C        Have each processor calculate local area
      CALL TRAP(ai, bi, N, H, AREAi)
C        Have processor 0 get ready to sum results and receive information
C        TAREA is total area which is sum of areas AREAi
      IF(ID .EQ. 0) THEN
      TAREA=AREAi
C        Get ready to receive information from other processors
      DO 100 ISOURCE=1,NP-1
      CALL MPI_RECV(AREAi, 1, MPI_REAL, ISOURCE, ITAG,
     1 MPI_COMM_WORLD, STATUS, IERR)
      TAREA=TAREA+Areai
100         CONTINUE
      ELSE
C        If processor ID greater than 0, then send results to processor 0
      CALL MPI_SEND(AREAi, 1, MPI_REAL, IDESTIN, ITAG,
    1 MPI_COMM_WORLD, IERR)
      END IF
C        Write out results
      IF(ID .EQ. 0) THEN
      WRITE(*,200) TAREA
200         FORMAT(1x,'AREA BY TRAPEZOIDAL RULE = ',E14.7)
      END IF
      CALL MPI_FINIALIZE(IERR)
      STOP
      END
```

■

Example 11-4.

The next example program gets a little more complicated. The program example4 is designed to test the transfer of an array using the MPI_SEND and MPI_RECV commands. The example assumes the use of 4 processors.

```
      PROGRAM EXAMPLE4
C        Assume 4 processors are being used-fill array with integer values 1-48
      DIMENSION A(3,4,4)
      INTEGER STATUS
      INCLUDE 'mpif.h'
      CALL MPI_INIT(IERR)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, ID, IERR)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NP, IERR)
C        Each processor fills in part of the array
      J=ID+1
      DO 20 I=1,3
      DO 21 K=1,4
      A(I,J,K)=I+3.*(J-1)+12*(K-1)
21       CONTINUE
20       CONTINUE
C        Define new array ANEWA describing blocks, block length, stride
       CALL MPI_TYPE_VECTOR(4, 3, 12, MPI_REAL, ANEWA, IERR)
C       You must commit this new array to all processors
      CALL MPI_TYPE_COMMIT(ANEWA, IERR)
C        Note in sending new array – only position of first element need be specified.
      IF(ID .NE. 0) THEN
      k=ID+1
      CALL MPI_SEND(A(1,k,1), 1, ANEWA, 0, 30, MPI_COMM_WORLD, IERR)
      ELSE
      DO 40 I=1,3
      K=I+1
      ISOURCE=I
      CALL MPI_RECV(A(1,K,1), 1, ANEWA, ISOURCE, 30,
     1 MPI_COMM_WORLD, STATUS, IERR)
40       CONTINUE
      END IF
      IF(ID .EQ. 0) THEN
C        Write out array and indexing
      OPEN(4,FILE='EX4.DAT',STATUS='UNKNOWN')
      DO 50 I=1,3
      DO 60 J=1,4
      DO 70 K=1,4
      INT=I+3*(J-1)+12*(K-1)
      WRITE(4,90)INT,I,J,K,A(I,J,K)
90       FORMAT(1x,4(1x,I3),1x,F5.1)
70       CONTINUE
60       CONTINUE
50       CONTINUE
      close (4)
      END IF
      CALL MPI_FINALIZE(IERR)
      STOP
      END
```