2 Simple MPI / FORTRAN Applications

2.1 Introduction

The numerical algorithms discussed throughout this textbook can be implemented by using Message Passing Interface (MPI) either in FORTRAN or C++ language. MPI/FORTRAN is essentially the same as the "regular" FORTRAN 77 and/or FORTRAN 90, with some additional, "special FORTRAN" statements for parallel computation, sending (or broadcasting), receiving and merging messages among different processors. Simple and most basic needs for these "special" FORTRAN statements can be explained and illustrated through two simple examples, which will be described in subsequent sections.

2.2 Computing Value of π by Integration

The numerical value of π ($\approx$3.1416) can be obtained by integrating the following function:

$$f(x)=\frac{4}{1+x^2},\quad \text{for } x=[0,1] \tag{2.1}$$

Since

$$\int \frac{1}{a+bx^2}dx=\frac{1}{\sqrt{ab}}\tan^{-1}\left(\frac{x\sqrt{ab}}{a}\right)$$

hence

$$\int_0^1 \frac{4}{1+x^2}dx=4\left[\tan^{-1}(x)\right]_0^1=\pi$$

Integrating the function given in Eq.(2.1) can be approximated by computing the area under the wave f(x), as illustrated in Figure 2.1.

In Figure 2.1, assuming the domain of interests, $x = 0 \rightarrow 1$, is divided into $n = 8$ segments. For example, segment 1 will correspond to $x = 0$ and 2/16, segment 4 corresponds to $x = 6/16$ and 8/16, etc. Thus, the integral size $h = (1-0)/8 = 1/8$. It is further assumed that the number of processors (= numprocs) to be used for parallel computation is 4 (= processors P_0, P_1, P_2, and P_3). The values of "myid" in Table 2.1 (see loop 20) are given by 0, 1, 2, and 3, respectively.

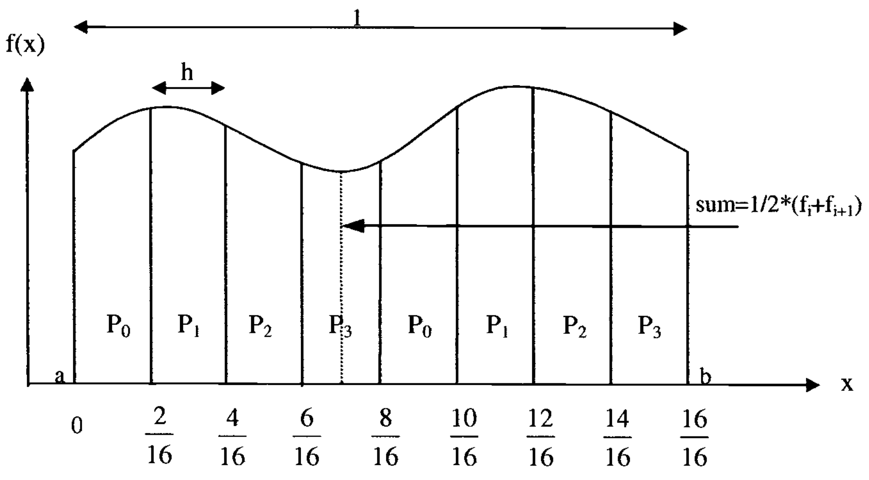


Figure 2.1 Computing π by Integrating $f(x) = \dfrac{4}{1+x^2}$

According to Figure 2.1, a typical processor (such as P_3) will be responsible to compute the areas of two segments. Thus, the total areas computed by processor P_3 are:

$$(\text{Area})_{P_3} = \frac{1}{2}\left(f\Big|_{x=\frac{6}{16}} + f\Big|_{x=\frac{8}{16}} \right) h + \frac{1}{2}\left(f\Big|_{x=\frac{14}{16}} + f\Big|_{x=\frac{16}{16}} \right) h \tag{2. 2}$$

$$= (\text{Area of segment 4}) + (\text{Area of segment of 8}) \tag{2. 3}$$

or

$$(\text{Area})_{P_3} = (h * \text{Average height of segment 4}) + (h * \text{Average height of segment 8}) \tag{2.4}$$

$$= h * (\text{Average height of segment 4} + \text{Average height of segment 8}) \tag{2. 5}$$

$$= h * \left\{ f\left(@\ x = \tfrac{7}{16}\right) + f\left(@\ x = \tfrac{15}{16}\right) \right\} \tag{2. 6}$$

$$(\text{Area})_{P_3} = h * \{\text{sum}\} = \text{mypi} \tag{2. 7}$$

The values of {sum} and mypi are computed inside and outside loop 20, respectively (see Table 2.1).

Since there are many comments in Table 2.1, only a few remarks are given in the following paragraphs to further clarify the MPI/FORTRAN code:

Remark 1. In a parallel computer environment, and assuming four processors (= P_0, P_1, P_2, and P_3) are used, then the same code (such as Table 2.1) will be executed by each and every processor.

Remark 2. In the beginning of the main parallel MPI/FORTRAN code, "special" MPI/FORTRAN statements, such as include "mpif.h", call MPI_INIT, ..., and call MPI_COMM_SIZE (or near the end of the main program, such as call MPI_FINALIZE), are always required.

Remark 3. The variable "myid" given in Table 2.1 can have integer values as 0, 1, 2, ... (# processors −1) = 3, since # processors = numprocs = 4 in this example.

Remark 4. The "If-Endif" block statements, shown near the bottom of Table 2.1, are only for printing purposes, and therefore, these block statements can be executed by any one of the given processors (such as by processor P_0 or myid = 0), or by processor P_3 (or myid = 3). To be on the safe side, however, myid = 0 is used, so that the parallel MPI code will not give an error message when it is executed by a single processor!

Remark 5. The parameters given inside the call MPI_BCAST statement have the following meanings:

1st parameter = n = The "information" that needs to broadcast to all processors. This "information" could be a scalar, a vector, or a matrix, and the number could be INTEGER, or REAL, etc.

In this example, n represents an INTEGER scalar.

2nd parameter = 1 (in this example), which represents the "size" of the 1st parameter.

3rd parameter = MPI_INTEGER, since in this example the 1st parameter is an integer variable.

4th parameter = 0, which indicates that the 1st parameter will be broadcasting to all other processors.

Remark 6. Depending on the processor number (= myid), each processor will be assigned to compute a different segment of rectangular areas (see do-loop 20, in Table 2.1).

Remark 7. After executing line "mypi=h*sum," each and every processor has already computed its own (or local) net areas (= area of 2 rectangular segments, as shown in Figure 2.1), and is stored under variable "mypi." Therefore, the final value for π can be computed by adding the results from each processor. This task is done by "call MPI_REDUCE." The important (and less obvious) parameters involved in " call MPI_REDUCE," shown in Table 2.1 are explained in the following paragraphs:

1st parameter = mypi = net (rectangular) areas, computed by each processor.

2nd parameter = pi = total areas (= approximate value for π), computed by adding (or summing) each processor's area.

3rd parameter = 1 (= size of the 1st parameter)

4th parameter = MPI_DOUBLE_PRECISION = double precision is used for summation calculation.

5th parameter = MPI_SUM = performing the "summing" operations from each individual processor's results (for some application, "merging" operations may be required).

6th parameter = 0

Remark 8. Before ending an MPI code, a statement, such as call MPI_FINALIZE (see Table 2.1), needs to be included.

Table 2.1 Compute π by Parallel Integration [2.1]

```
c  pi.f - compute pi by integrating f(x) = 4/(1 + x**2)
c
c  Each node:
c   1) receives the number of rectangles used in the approximation.
c   2) calculates the areas of it's rectangles.
c   3) Synchronizes for a global summation.
c  Node 0 prints the result.
c
c  Variables:
c
c   pi  the calculated result
c   n   number of points of integration.
c   x        midpoint of each rectangle's interval
c   f        function to integrate
c   sum,pi     area of rectangles
c   tmp        temporary scratch space for global summation
c   i        do loop index
c++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      program main
      include 'mpif.h'
      double precision  PI25DT
      parameter       (PI25DT = 3.141592653589793238462643d0)
      double precision  mypi, pi, h, sum, x, f, a,time1,time2
      integer n, myid, numprocs, i, rc
c                           function to integrate
      f(a) = 4.d0 / (1.d0 + a*a)
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      print *, 'Process ', myid, ' of ', numprocs, ' is alive'
c-----------------------------------------------------------------------
       time1=MPI_Wtime()
c-----------------------------------------------------------------------
      sizetype  = 1
      sumtype   = 2
 10   if ( myid .eq. 0 ) then
c        write(6,98)
c 98     format('Enter the number of intervals: (0 quits)')
c        read(5,99) n
c 99     format(i10)
      read(101,*) n
c     n=10000000
      endif
      call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
c     check for quit signal
      if ( n .le. 0 ) goto 30
c     calculate the interval size
      h = 1.0d0/n
      sum  = 0.0d0
      do 20 i = myid+1, n, numprocs
        x = h * (dble(i) - 0.5d0)
        sum = sum + f(x)
 20   continue
      mypi = h * sum
c     collect all the partial sums
      call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
     $     MPI_COMM_WORLD,ierr)
c     node 0 prints the answer.
      if (myid .eq. 0) then
        write(6, 97) pi, abs(pi - PI25DT)
 97     format('  pi is approximately: ', F18.16,
     +          '  Error is: ', F18.16)
      endif
c     goto 10
 30   call MPI_FINALIZE(rc)
c-----------------------------------------------------------------------
      time2=MPI_Wtime()
      write(6,*) 'MPI elapse time= ',time2-time1
c-----------------------------------------------------------------------
      stop
      end
```

2.3 Matrix-Matrix Multiplication

The objective of this section is to again illustrate the usage of few basic parallel MPI/FORTRAN statements for simple applications such as multiplying 2 matrices. To further simplify the discussion, it is assumed that 4 processors (numtasks = number of processors = 4, see line #021 in Table 2.2) are used for parallel multiplication of 2 square matrices A (1000, 1000) and B (1000, 1000). The main steps involved in this example are summarized as follows:

<u>Step 1:</u> Input (or generate) matrices [A] and [B] by a master program (= processor 0). This task is done by lines # 048 – 058 in Table 2.2.

<u>Step 2:</u> The master processor will distribute certain columns of matrix [B] to its workers (= numworkers = numtasks – 1 = 4 - 1 = 3 workers, see line #022 in Table 2.2). Let NCB = Number of Columns of [B] = 1,000. The average number of columns to be received by each "worker" processor (from a "master" processor) is:

Avecol = NCB/numworkers = 1,000/3 = 333 (see line #060).

The "extra" column(s) to be taken care of by certain "worker" processor(s) is calculated as:

Extra = mod (NCB, numworkers) = 1 (see line #061).

In this example, we have a "master" processor (taskid = 0, see line #020) and 3 "worker" processors (taskid = 1, 2, or 3). Each "worker" processor has to carry a minimum of 333 columns of [B]. However, those worker processor number(s) that are less than or equal to "extra" (= 1), such as worker processor 1, will carry:

Cols = avecol + 1 = 334 columns (see line #066).

The parameter "offset" is introduced and initialized to 1 (see line #062). The value of "offset" is used to identify the "starting column number" in the matrix [B] that will be sent (by a MASTER processor) to each "worker" processor. Its value is updated as shown at line #079. For this particular example, the values of "offset" are 1, 335, and 668 for "worker processor" number 1, 2, and 3 (or processor numbers P_1, P_2, and P_3), respectively.

In other words, worker processor P_1 will receive and store 334 columns of [B], starting from column 1 (or offset = 1), whereas worker processors P_2 (and P_3) will receive and store 333 columns of [B], starting from column 335 (and 668), respectively.

Inside loop 50 (see lines #064 – 080), the "master" processor (P_0) will send the following pieces of information to the appropriate "worker processors":

(a) The value of "offset" (see line #071)
(a) The number of columns of [B] (see line #073)
(b) The entire matrix [A] (see line #075)
(c) The specific columns of [B] (see line #077)

The parameters involved in MPI_ SEND are explained as follows:

1st parameter = name of the variable (could be a scalar, a vector, a matrix, etc.) that needs to be sent
2nd parameter = the size (or dimension) of the 1st parameter (expressed in words, or numbers)
3rd parameter = the type of variable (integer, real, double precision, etc.)
4th parameter = dest = destination, which processor will receive the message (or information)
5th parameter = mtype = 1 (= FORM_MASTER, also see lines #013, 063)
6th parameter = as is, unimportant
7th parameter = as is, unimportant

Having sent all necessary data to "worker processors," the "master processor" will wait to receive the results from its workers (see lines # 081 – 091), and print the results (see lines # 092 – 105).

Step 3: Tasks to be done by each "worker" processor.

(a) Receive data from the "master" processor (see lines #106 – 118).
(b) Compute the matrix-matrix multiplications for [entire A]*[portion of columns B] ≡ [portion of columns C], see lines #119 – 125.
(c) Send the results (= [portion of columns C]) to the "master" processor (see lines #126 – 136).

It should be noted here that the parameters involved in MPI_RCV are quite similar (nearly identical to the ones used in MPI_SEND).

Table 2.2 Parallel Computation of Matrix Times Matrix [2.1]

```
      program mm                                                    ! 001
c---------------------------------------------------                ! 002
c.This file is stored under ~/cee/MPI-oduSUNS/matrix_times_matrix_mpi.f ! 003
c-----------------------------------------------------------------  ! 004
       include 'mpif.h'                                             ! 005
c      NRA : number of rows in matrix A                             ! 006
c      NCA : number of columns in matrix A                          ! 007
c      NCB : number of columns in matrix B                          ! 008
       parameter (NRA = 1000)                                       ! 009
      parameter (NCA = 1000)
      parameter (NCB = 1000)
      parameter (MASTER = 0)
      parameter (FROM_MASTER = 1)
      parameter (FROM_WORKER = 2)
      integer         numtasks,taskid,numworkers,source,dest,mtype,
     &           cols,avecol,extra, offset,i,j,k,ierr
      integer status(MPI_STATUS_SIZE)
      real*8a(NRA,NCA), b(NCA,NCB), c(NRA,NCB)
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, taskid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numtasks, ierr )
      numworkers = numtasks-1
      print *, 'task ID= ',taskid
c++++++++++++++++++++++++++++++++++++++++++++++
c      write(6,*) 'task id = ',taskid
c      call system ('hostname')
c      ! to find out WHICH computers run this job
c      ! this command will SLOW DOWN (and make UNBALANCED
c      ! workloads amongst processors)
c++++++++++++++++++++++++++++++++++++++++++++++
      if(numworkers.eq.0) then
      time00 =  MPI_WTIME()
C     Do matrix multiply
        do 11 k=1, NCB
          do 11 i=1, NRA
            c(i,k) = 0.0
            do 11 j=1, NCA
              c(i,k) = c(i,k) + a(i,j) * b(j,k)
 11     continue
       time01 =  MPI_WTIME()
        write(*,*) ' C(1,1)    : ', c(1,1)
        write(*,*) ' C(nra,ncb): ',C(nra,ncb)
        write(*,*)
        write(*,*) ' Time  me=0: ', time01 - time00
        go to 99
       endif
C ************************* master task *************************
      if (taskid .eq. MASTER) then
      time00 =  MPI_WTIME()
C     Initialize A and B
        do 30 i=1, NRA
          do 30 j=1, NCA
          a(i,j) = (i-1)+(j-1)
 30     continue
        do 40 i=1, NCA
          do 40 j=1, NCB
            b(i,j) = (i-1)*(j-1)
 40      continue
C     Send matrix data to the worker tasks
        avecol = NCB/numworkers
        extra = mod(NCB, numworkers)
        offset = 1
      mtype = FROM_MASTER                                                     ! 063
      do 50 dest=1, numworkers                                                ! 064
        if (dest .le. extra) then                                             ! 065
          cols = avecol + 1                                                   ! 066
        else                                                                  ! 067
          cols = avecol                                                       ! 068
        endif                                                                 ! 069
        write(*,*)'   sending',cols,' cols to task',dest                      ! 070
        call MPI_SEND( offset, 1, MPI_INTEGER, dest, mtype,                   ! 071
     &                 MPI_COMM_WORLD, ierr )                                 ! 072
        call MPI_SEND( cols, 1, MPI_INTEGER, dest, mtype,                     ! 073
     &                 MPI_COMM_WORLD, ierr )                                 ! 074
        call MPI_SEND( a, NRA*NCA, MPI_DOUBLE_PRECISION, dest, mtype,         ! 075
     &                 MPI_COMM_WORLD, ierr )                                 ! 076
        call MPI_SEND( b(1,offset), cols*NCA, MPI_DOUBLE_PRECISION,           ! 077
     &                 dest, mtype, MPI_COMM_WORLD, ierr )                    ! 078
        offset = offset + cols                                                ! 079
 50   continue                                                                ! 080
C     Receive results from worker tasks                                       ! 081
      mtype = FROM_WORKER                                                     ! 082
      do 60 i=1, numworkers                                                   ! 083
        source = i                                                            ! 084
        call MPI_RECV( offset, 1, MPI_INTEGER, source,                        ! 085
     &                 mtype, MPI_COMM_WORLD, status, ierr )                  ! 086
        call MPI_RECV( cols, 1, MPI_INTEGER, source,                          ! 087
     &                 mtype, MPI_COMM_WORLD, status, ierr )                  ! 088
        call MPI_RECV( c(1,offset), cols*NRA, MPI_DOUBLE_PRECISION,           ! 089
     &                 source, mtype, MPI_COMM_WORLD, status, ierr )          ! 090
 60   continue                                                                ! 091
      time01  = MPI_WTIME()                                                   ! 092
C     Print results                                                           ! 093
c      do 90 i=1, NRA                                                         ! 094
c        do 80 j = 1, NCB                                                     ! 095
c          write(*,70)c(i,j)                                                  ! 096
c 70       format(2x,f8.2,$)                                                  ! 097
c 80     continue                                                             ! 098
c        print *, ' '                                                         ! 099
c 90   continue                                                               ! 100
      write(*,*) ' C(1,1)    : ', c(1,1)                                      ! 101
      write(*,*) ' C(nra,ncb): ',C(nra,ncb)                                   ! 102
      write(*,*)                                                              ! 103
      write(*,*) ' Time  me=0: ', time01 - time00                             ! 104
    endif                                                                     ! 105
C ************************** worker task ****************************       ! 106
      if (taskid.gt.MASTER) then                                              ! 107
      time11  = MPI_WTIME()                                                   ! 108
C     Receive matrix data from master task                                    ! 109
      mtype = FROM_MASTER                                                     ! 110
      call MPI_RECV( offset, 1, MPI_INTEGER, MASTER,                          ! 111
     &               mtype, MPI_COMM_WORLD, status, ierr )                    ! 112
      call MPI_RECV( cols, 1, MPI_INTEGER, MASTER,                            ! 113
     &               mtype, MPI_COMM_WORLD, status, ierr )                    ! 114
      call MPI_RECV( a, NRA*NCA, MPI_DOUBLE_PRECISION, MASTER,                ! 115
     &              mtype, MPI_COMM_WORLD, status, ierr )
       call MPI_RECV( b, cols*NCA, MPI_DOUBLE_PRECISION, MASTER,
     &              mtype, MPI_COMM_WORLD, status, ierr )
C      Do matrix multiply
       do 100 k=1, cols
         do 100 i=1, NRA
           c(i,k) = 0.0
           do 100 j=1, NCA
             c(i,k) = c(i,k) + a(i,j) * b(j,k)
 100   continue
C      Send results back to master task
       mtype = FROM_WORKER
       call MPI_SEND( offset, 1, MPI_INTEGER, MASTER, mtype,
     &              MPI_COMM_WORLD, ierr )
       call MPI_SEND( cols, 1, MPI_INTEGER, MASTER, mtype,
     &              MPI_COMM_WORLD, ierr )
       call MPI_SEND( c, cols*NRA, MPI_DOUBLE_PRECISION, MASTER,
     &               mtype, MPI_COMM_WORLD, ierr )
      time22 =  MPI_WTIME()
      write(*,*) ' Time  me=',taskid,': ', time22 - time11
     endif
 99  call MPI_FINALIZE(ierr)
     end
```

2.4 MPI Parallel I/O

Some basics in Input/Output (I/O) operations under MPI environments are explained and demonstrated in Table 2.3

Table 2.3 I/O Operations under MPI Environments

```
      program mpiio
      include 'mpif.h'
c       implicit real*8(a-h,o-z)     ! will get error if use this "implicit" stmt
c***********************************************************************
c       Purposes:  Using MPI parallel i/o for writing & reading by different
c                  processors, on "different segments" of the "same" file
c       Person(s): Todd and Duc T. Nguyen
c       Latest Date:  April 10, 2003
c       Stored at:    cd ~/cee/mpi_sparse_fem_dd/mpi_io.f
c       Compile ??:   Just type
c                     tmf90 -fast -xarch=v9 mpi_io.f -lmpi
c       Execute ??:   Just type (assuming 2 processors are used)
c                     bsub -q hpc-mpi-short -I -n 2 a.out
c       Output:       Stored at files 301+(me= processors 0, 1, etc...)
c***********************************************************************
      real*8 a, b
      real*8 t1,t2,t3
      parameter (nns = 10000000)
      integer op,nns,nsz,myfh
      dimension a(nns),b(nns)
      integer (kind=mpi_offset_kind) idisp
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, me, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
      ns = nns/np
      ip = 101
      op = 301+me
c      kind = MPI_OFFSET_KIND
c+++++++++++++++++++++++++++++++++++
c Initial testing arrays
c+++++++++++++++++++++++++++++++++++
      do i = 1,ns
        a(i) = i+ns*me
        b(i) = 0
      enddo
c+++++++++++++++++++++++++++++++++++
c Open the file
c+++++++++++++++++++++++++++++++++++
      call MPI_FILE_OPEN(MPI_COMM_WORLD, 'test',
     & MPI_MODE_rdwr+mpi_mode_create,MPI_INFO_NULL,myfh,ierr)
c+++++++++++++++++++++++++++++++++++
c Set view point for each process
c+++++++++++++++++++++++++++++++++++
      idisp = ns*8*me
      write(op,*) '---------------------'
      write(op,*) 'me,kind',me,kind
      write(op,*) 'idisp',idisp
      call MPI_FILE_SET_VIEW(myfh,idisp,MPI_double_precision,
     & MPI_double_precision,'native',MPI_INFO_NULL,ierr )
c      call mpi_file_seek(myfh,idisp,mpi_seek_set,ierr)
      nsz = ns
      write(op,*) 'nsz',nsz
c+++++++++++++++++++++++++++++++++++
c Write array to the disk
c+++++++++++++++++++++++++++++++++++
c      write(op,*) 'a',(a(i),i=1,nsz)
      t1 = MPI_WTIME()
      call MPI_FILE_WRITE(myfh,a,nsz,MPI_double_precision,
     & mpi_status_ignore,ierr)
      t2 = MPI_WTIME()
      write(op,*) 'Time to write(MPI) to the disk',t2-t1
c+++++++++++++++++++++++++++++++++++
c Read file
c+++++++++++++++++++++++++++++++++++
      idisp = ns*8*(me+1)
      if (me .eq. np-1) then
        idisp = 0
      endif
      call MPI_FILE_SET_VIEW(myfh,idisp,MPI_double_precision,
     & MPI_double_precision,'native',MPI_INFO_NULL,ierr )
      call MPI_FILE_READ(myfh,b,nsz,MPI_double_precision,
     & mpi_status_ignore,ierr)
      call MPI_FILE_CLOSE(myfh,ierr)
      t3 = MPI_WTIME()
      write(op,*) 'Time to read(MPI) from the disk',t3-t2
c     write(op,*) 'b',(b(i),i=1,nsz)
c     call MPI_FILE_CLOSE(myfh,ierr)
999   call MPI_FINALIZE(ierr)
      stop
      end
```

Preliminary timing for READ/WRITE by a different number of processors is reported in Table 2.4.

Table 2.4 Preliminary Performance of MPI Parallel I/O

(Using HELIOS, but compiled and executed from CANCUN, SUN_10,000 computer)

NP	1	2	4	Comments
Write	83.04 sec	46.34 sec (me = 0)	29.75 sec	using MPI wall clock time subroutine
			30.02 sec	
		46.68 sec (me = 1)	37.55 sec	write is slower than read
			58.38 sec	
Read	15.39 sec	18.34 sec (me = 0)	11.77 sec	Parallel write seems to offer some speed
			28.35 sec	
		19.68 sec (me = 1)	22.36 sec	
			8.79 sec	Parallel read seems to be worse
	Each processor write 10 million double procession words, and read 10 Million words	Each processor write and read 5 million double precision words (in parallel)	Correspond to 4 processors $\text{me} = \begin{cases} 0 \\ 1 \\ 2 \\ 3 \end{cases}$ each processor write and read $2.5*10^6$ words	

2.5 Unrolling Techniques [1,9]

To enhance the performance of the codes, especially on a single vector processor, unrolling techniques are often used to reduce the number of "load and store" movements (to and from the CPU), and, therefore, code performance can be improved (up to a factor of 2, on certain computer platforms).

To illustrate the main ideas, consider the simple task of finding the product of a given matrix [A] and a vector {x}, where

$$[A]=\begin{bmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\end{bmatrix} \text{ and } x=\begin{Bmatrix}2\\0\\1\\1\end{Bmatrix} \tag{2.8}$$

Algorithm 1: Dot product operations

In this algorithm, each row of a matrix [A] operates on a vector {x} to obtain the "final" solution. The final answer for $b_1=\begin{Bmatrix}1\\2\\3\\4\end{Bmatrix}\cdot\begin{Bmatrix}2\\0\\1\\1\end{Bmatrix}=9$, which involves the dot product operations of two vectors. The FORTRAN code for this algorithm is given in Table 2.5.

Table 2.5 Matrix Times Vector (Dot Product Operations)

```
      Do 1 I = 1, N (say = 4)
      Do 1 J = 1, N
1       b(I) = b(I) + A(I, J) * x(J)
```

Algorithm 2: Dot product operations with unrolling techniques

We can group a few (say NUNROL = 2) rows of [A] and operate on a vector {x}. Thus, algorithm 1 can be modified and is given in Table 2.6.

Table 2.6 Matrix Times Vector (Dot Product Operations, with Unrolling Level 2)

```
      Do 1 I = 1, N, NUNROL (= 2)
      Do 1 J =1, N
        b(I) = b(I) + A(I, J) * x(J)
1       b(I+1) = b(I+1) + A(I+1, J) * x(J)
```

Algorithm 3: Saxpy operations

In this algorithm, each column of matrix [A] operates on an appropriate component of vector {x} to get a "partial, or incomplete" solution vector {b}. Thus, the first "partial" solution for {b} is given as:

$$\{b\}_{\text{incomplete}} = \begin{Bmatrix} 1 \\ 5 \\ 9 \\ 13 \end{Bmatrix} * 2 = \begin{Bmatrix} 2 \\ 10 \\ 18 \\ 26 \end{Bmatrix} \tag{2.9}$$

and eventually, the "final, complete" solution for {b} is given as:

$$\{b\}_{\text{final}} = \begin{Bmatrix} 2 \\ 10 \\ 18 \\ 26 \end{Bmatrix} + \begin{Bmatrix} 2 \\ 6 \\ 10 \\ 14 \end{Bmatrix} * (0) + \begin{Bmatrix} 3 \\ 7 \\ 11 \\ 15 \end{Bmatrix} * (1) + \begin{Bmatrix} 4 \\ 8 \\ 12 \\ 16 \end{Bmatrix} * (1) = \begin{Bmatrix} 9 \\ 25 \\ 41 \\ 61 \end{Bmatrix} \tag{2.10}$$

The FORTRAN code for this algorithm is given in Table 2.7.

Table 2.7 Matrix Times Vector (Saxpy Operations)

```
      Do 1 I = 1, N
      Do 1 J = 1, N
        b(J) = b(J) + A(J, I) * x(I)
1       Continue
```

The operation inside loop 1 of Table 2.7 involves a summation of a scalar a (= x (I), in this case, since x(I) is independent from the innermost loop index J) times a vector x (= A(J, I), in this case, since the Ith column of [A] can be considered as a vector) plus a previous vector y (= b(J), in this case), hence the name SAXPY!

Algorithm 4: Saxpy operations with unrolling techniques

We can group a few (say NUNROL = 2) columns of matrix [A] and operate on appropriate components of vector {x}. Thus, algorithm 3 can be modified as shown in Table 2.8.

Table 2.8 Matrix Times Vector (Saxpy Operation with Unrolling Level 2)

```
      Do 1 I = 1, N, NUNROL (= 2)
      Do 1 J = 1, N
        b(J) = b(J) + A(J, I) * x(I) + A(J, I+1) * x(I+1)
1       Continue
```

<u>Remarks:</u> Unrolling dot product operations are different from unrolling saxpy operations in several aspects:

(a) The former (unrolling dot product) gives a "final, or complete" solution, whereas the latter (unrolling saxpy) gives a "partial, or incomplete" solution.
(b) In FORTRAN coding, the former requires "several" FORTRAN statements inside the innermost loop (see Table 2.6), whereas the latter requires "one long" FORTRAN statement (see Table 2.8).
(c) The former involves the operation of the dot product of two given vectors, whereas the latter involves the operation of a CONSTANT times a VECTOR, plus another vector.
(d) On certain computer platforms (such as Cray 2, Cray YMP and Cray – C90, etc.), saxpy operations are faster than dot product operations. Thus, algorithms and/or solution strategies can be tailored to specific computer platforms to improve the numerical performance.
(e) In Tables 2.6 and 2.8, if N is "not" a multiple of "NUNROL", then the algorithms have to be modified slightly in order to take care of a few remaining rows (or columns) of matrix [A], as can be seen in Exercises 2.4 and 2.6.

2.6 Parallel Dense Equation Solvers

In this section, it will be demonstrated that simple, yet highly efficient parallel strategies can be developed for solving a system of dense, symmetrical, positive definite equations. These strategies are based upon an efficient matrix times matrix subroutine, which also utilizes the "unrolling" techniques discussed in the previous sections.

2.6.1 Basic Symmetrical Equation Solver

Systems of linear, symmetrical equations can be represented as:

$$A \cdot x = b \tag{2.11}$$

One way to solve Eq.(2.11) is to first decompose the coefficient matrix A into the product of two triangular matrices

$$A = U^T U \tag{2.12}$$

Where U is an upper-triangular matrix which can be obtained by

when $i \neq j$, then
$$u_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj}}{u_{ii}} \tag{2.13}$$

when $i = j$, then
$$u_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} u_{ki}^2} \tag{2.14}$$

Then the unknown vector x can be solved through the forward/backward elimination, such as:

$$U^T y = b \tag{2.15}$$

for y, with

$$y_j = \frac{b_j - \sum_{i=1}^{j-1} u_{ij} y_i}{u_{jj}} \tag{2.16}$$

and to solve

$$Ux = y \tag{2.17}$$

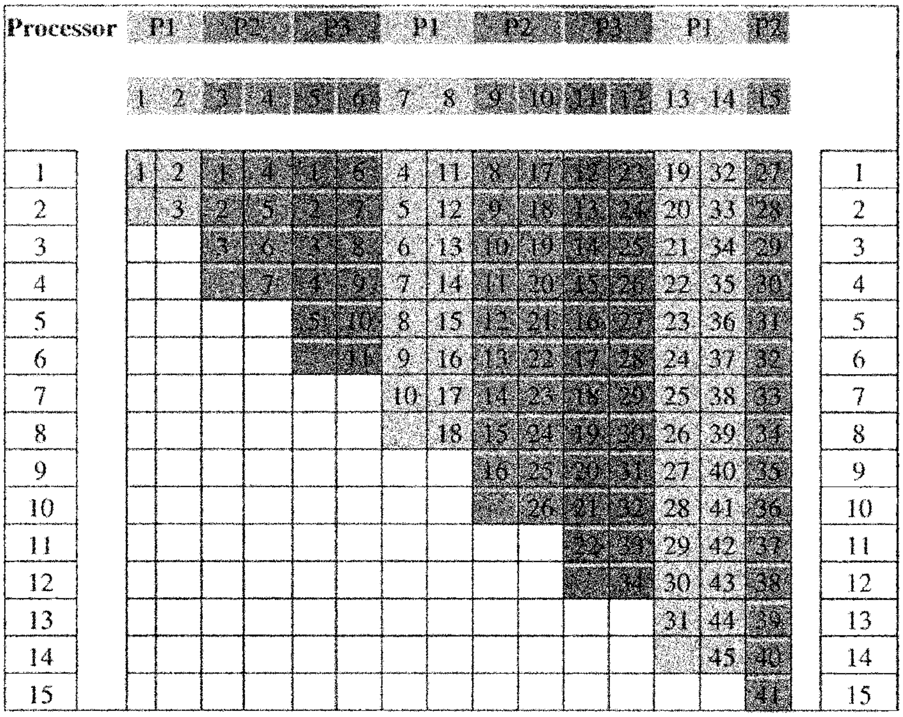
for x, with

$$x_j = \frac{y_j - \sum_{i=j+1}^{n} u_{ji} x_i}{u_{jj}} \tag{2.18}$$

The efficiency of an equation solver on massively parallel computers with distributed memory is dependent on both its vector and/or cache performance and its communication performance. In this study, we have decided to adopt a skyline column storage scheme to exploit dot product operations.

2.6.2 Parallel Data Storage Scheme

Assuming 3 processors are used to solve a system of 15 matrix equations, the matrix will be divided into several parts with 2 columns per blocks (ncb = 2). Therefore, from Figure 2.2, processor 1 will handle columns 1, 2, 7, 8, 13, and 14. Also, processor 2 will handle columns 3, 4, 9, 10, and 15, and processor 3 will handle columns 5, 6, 11, and 12. The columns that belong to each processor will be stored in a one-dimensional array in a column-by-column fashion. For example, the data in row 4 and column 7 will be stored by processor 1 at the 7th location of one-dimensional array A of processor 1. Likewise, each processor will store only portions of the whole matrix [A]. The advantage of this storage scheme is that the algorithm can solve a much bigger problem.



Processor	P1		P2		P3		P1		P2		P3		P1		P2	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
1	1	2	1	4	1	6	4	11	8	17	12	23	19	32	27	1
2		3	2	5	2	7	5	12	9	18	13	24	20	33	28	2
3			3	6	3	8	6	13	10	19	14	25	21	34	29	3
4				7	4	9	7	14	11	20	15	26	22	35	30	4
5					5	10	8	15	12	21	16	27	23	36	31	5
6						11	9	16	13	22	17	28	24	37	32	6
7							10	17	14	23	18	29	25	38	33	7
8								18	15	24	19	30	26	39	34	8
9									16	25	20	31	27	40	35	9
10										26	21	32	28	41	36	10
11											22	33	29	42	37	11
12												34	30	43	38	12
13													31	44	39	13
14														45	40	14
15															41	15

Figure 2.2 Block Columns Storage Scheme

Two more small arrays are also needed to store the starting columns (icolst) and ending columns (icolend) of each block.

Therefore, from this example:

$$\text{icolst}_1 := \begin{pmatrix} 1 \\ 7 \\ 13 \end{pmatrix} \qquad \text{icolend}_1 := \begin{pmatrix} 2 \\ 8 \\ 14 \end{pmatrix}$$

$$\text{icolst}_2 := \begin{pmatrix} 3 \\ 9 \\ 15 \end{pmatrix} \qquad \text{icolend}_2 := \begin{pmatrix} 4 \\ 10 \\ 15 \end{pmatrix}$$

$$\text{icolst}_3 := \begin{pmatrix} 5 \\ 11 \end{pmatrix} \qquad \text{icolend}_3 := \begin{pmatrix} 6 \\ 12 \end{pmatrix}$$

It should be noted that sizes of arrays {icolst} and {icolend} are the same for the same processor, but may be different from other processors. In fact, it depends on how many blocks are assigned to each processor (noblk). For this example, processors 1 and 2 each stores 3 blocks of the matrix, and processor 3 stores 2 blocks of the matrix.

2.6.3 Data Generating Subroutine

The stiffness matrix, stored in a one-dimensional array {A} and the right-hand-side load vector {b} will be automatically generated such that the solution vector {x} will be 1 for all values of {x}. Also, from Eq.2.14, the diagonal values of the stiffness matrix should be large to avoid the negative values in the square root operation. The general formulas to generate the stiffness matrix are:

$$a_{i,i} = 50000 \cdot (i+i)$$

$$a_{i,j} = i+j$$

and the formula for RHS vector can be given as:

$$b_i = \sum_{j=1}^{n} a_{i,j}\,.$$

2.6.4 Parallel Choleski Factorization [1.9]

Assuming the first four rows of the matrix A (see Figure 2.2) have already been updated by multiple processors, and row 5 is currently being updated, according to Figure 2.2, terms such as $u_{5,5}$... $u_{5,6}$ and $u_{5,11}$... $u_{5,12}$ are processed by processor 3. Similarly, terms such as $u_{5,7}$... $u_{5,8}$ and $u_{5,13}$... $u_{5,14}$ are handled by processor 1 while terms such as $u_{5,9}$, $u_{5,10}$ and $u_{5,15}$ are executed by processor 2.

As soon as processor 3 completely updated column 5 (or more precisely, updated the diagonal term $u_{5,5}$ since the terms $u_{1,5}$ $u_{2,5}$ ... $u_{4,5}$ have already been factorized earlier), it will send the entire column 5 (including its diagonal term) to all other processors. Then processor 3 will continue to update its other terms of row 5. At the same time, as soon as processors 1 and 2 receive column 5 (from processor 3), these processors will immediately update their own terms of row 5.

To enhance the computational speed, by using the optimum available cache, the "scalar" product operations (such as u_{ki}*u_{kj}) involved in Eq.(2.13) can be replaced by "sub-matrix" product operations. Thus, Eq.(2.13) can be re-written as:

$$[u_{ij}] = [u_{ii}]^{-1} \cdot \left([a_{ij}] - \sum_{k=1}^{i-1} [u_{ki}]^T [u_{kj}] \right) \qquad (2.19)$$

Similar "sub-matrix" expressions can be used for Eqs.(2.14, 2.16, 2.18).

2.6.5 A Blocked And Cache Based Optimized Matrix-Matrix Multiplication

Let's consider matrix C(m,n) to be the product of two dense matrices A and B of dimension (m,l) and (l,n), respectively.

$$C = A \cdot B \tag{2.20}$$

A basic matrix-matrix multiplication algorithm consists of triple-nested do loops as follows:

```
Do i=1,m
  Do j=1,n
    Do k=1,l
      c(i,j)=c(i,j)+a(i,k)*b(k,j)
    ENDDO
  ENDDO
ENDDO
```

Further optimization can be applied on this basic algorithm to improve the performance of the multiplication of two dense matrices. The following optimization techniques are used for the matrix-matrix multiplication sub-routine.

- Re-ordering of loop indexes
- Blocking and strip mining
- Loop unrolling on the considered sub-matrices
- Stride minimization
- Use of temporary array and leftover computations

2.6.5.1 Loop Indexes And Temporary Array Usage

Fortran uses column-major order for array allocation. In order to get a "stride" of one on most of the matrices involved in the triple do loops of the matrix-matrix multiplication, one needs to interchange the indices as follows:

```
Do j=1,n
  Do i=1,m
    Do k=1,l
      c(i,j)=c(i,j)+a(i,k)*b(k,j)
    ENDDO
  ENDDO
ENDDO
```

Note that in the above equation, the order of the index loop has been re-arranged to get a stride of 1 on matrix C and B. However, the stride on matrix A is m. In order to have a stride of 1 during the computation on matrix A, a temporary array is used to load a portion of matrix A before computation.

2.6.5.2 Blocking and Strip Mining

Blocking is a technique to reduce cache misses in nested array processing by calculating in blocks or strips small enough to fit in the cache. The general idea is that an array element brought in should be processed as fully as possible before it is flushed out.

To use the blocking technique in this basic matrix-matrix multiplication, the algorithm can be re-written by adding three extra outer loops (JJ, II, and KK) with an increment equal to the size of the blocks.

```
Do JJ=1,n,jb
  Do II=1,m,ib
    Do KK=1,l,kb
      Do J=JJ,min(n,JJ+jb-1)
        Do I=II,min(m,II+ib-1)
          Do K=KK,min(l,KK+kb-1)
            c(i,j)=c(i,j)+a(i,k)*b(k,j)
          ENDDO
        ENDDO
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

In the above code-segment, ib, jb, and kb are the block sizes for i, j, and k, respectively, and they are estimated function of the available cache size for different computer platforms.

2.6.5.3 Unrolling of Loops

Unrolling of loops is considered at various stages. To illustrate the idea, let's consider the inner do loop (see the index k) where the actual multiplication is performed. Here a 4 by 4 sub-matrix is considered at the time, and the inner do loop of Eqs.(2.13 – 2.14) can be unrolled in the following fashion:

```
DO j=1, number of J blocks
  DO i=1, number of I blocks
    DO k=1, number of K blocks
      C{i,j} = C{i,j} + T{1,1} * B{k,j}
      C{i+1,j} = C{i+1,j} + T{1,2} * B{k,j}
      C{i,j+1} = C{i,j+1} + T{1,1} * B{k,j+1}
```

```
            C{i+1,j+1} = C{i+1,j+1} + T{1,2} * B{k,j+1}
            C{i,j+2} = C{i,j+2} + T{1,1} * B{k,j+2}
            C{i+1,j+2} = C{i+1,j+2} + T{1,2} * B{k,j+2},
            C{i,j+3} = C{i,j+3} + T{1,1} * B{k,j+3}
            C{i+1,j+3} = C{i+1,j+3} + T{1,2} * B{k,j+3}
            C{i+2,j} = C{i+2,j} + T{1,1} * B{k,j}
            C{i+3,j} = C{i+3,j} + T{1,2} * B{k,j}
            C{i+2,j+1} = C{i+2,j+1} + T{1,1} * B{k,j+1}
            C{i+3,j+1} = C{i+3,j+1} + T{1,2} * B{k,j+1}
            C{i+2,j+2} = C{i+2,j+2} + T{1,1} * B{k,j+2}
            C{i+3,j+2} = C{i+3,j+2} + T{1,2} * B{k,j+2}
            C{i+2,j+3} = C{i+2,j+3} + T{1,1} * B{k,j+3}
            C{i+3,j+3} = C{i+3,j+3} + T{1,2} * B{k,j+3}
        ENDDO
      ENDDO
    ENDDO
```

In the above code-segment, T is the temporary 2-D array, which contains portions of the matrix [A].

2.6.6 Parallel "Block" Factorization

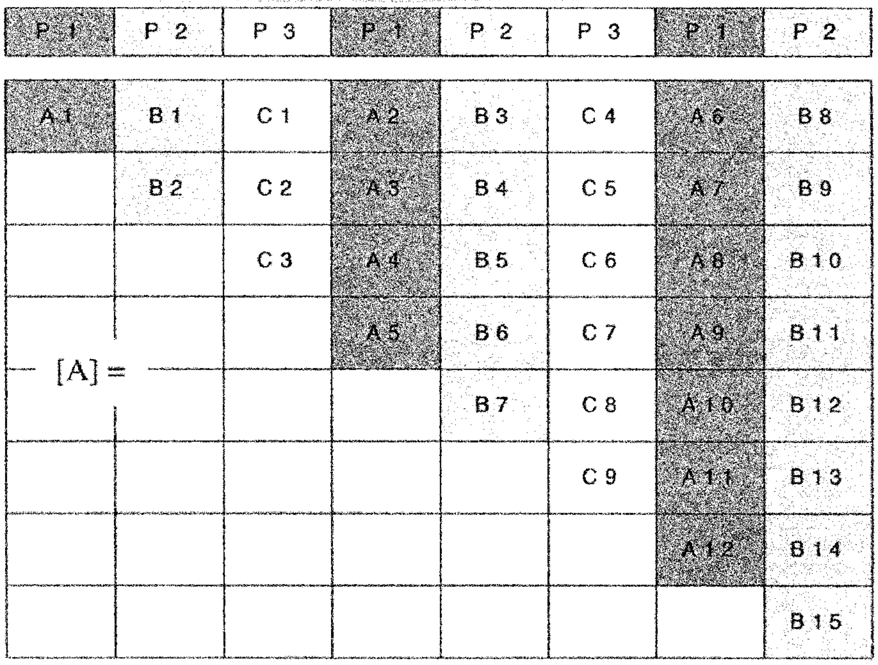


Figure 2.3 Parallel Block Factorization

The procedure starts at factorizing the first block of processor 1 by using basic sequential Choleski factorization. Then, processor 1 will send its first block (A1) to processor 2 and 3. After that, processor 1 will factorize A2 and A6 blocks. At the same time, processor 2 is factorizing B1, B3, and B8 blocks, and processor 3 is factorizing C1 and C4 blocks. Up to now, the first ncb rows are already factorized.

Note:

Suppose the block A2 is being factorized, the formula that can be used for factorization is

$$[A2]_{new} = ([A1]^T)^{-1} [A2]_{old}$$

Pre-multiply both side with $[A1]^T$, we get

$$[A1]^T [A2]_{new} = [A2]_{old}$$

$[A2]_{new}$, therefore, can be solved by performing the forward operations for ncb times.

After processor 2 finished factorizing blocks B1, B3, and B8, the block B2 will be the next block to be factorized. The numerator of equation (2.13) can be "symbolically" rewritten in the form of sub-matrix as:

$$[B2]_{new} = \sqrt{[B2]_{old} - [B1]^T [B1]} \tag{2.21}$$

"Squared" both sides of Eq.(6.21), one gets:

$$[B2]^T_{new} [B2]_{new} = [B2]_{old} - [B1]^T [B1] \tag{2.22}$$

Since the right-hand-side matrix of Eq.(2.22) is known, the "standard" Choleski method can be used to compute $[B2]_{new}$. After processor 2 finishes factorizing B2, it will send the information of this block to processor 3 and processor 1, respectively. Then, it will factorize block B4 and B9 while processor 3 and 1 are factorizing C2, C5, A3, and A7. Up to now, the first 2 (ncb) rows are already factorized.

Note:

Suppose block C2 is being factorized, the numerator of equation 3 can be re-written in sub-matrix form as:

$$[C2]_{new} = [C2]_{old} - [B1]^T [C1]$$

Then, the denominator term will be processed like the step that has been used to factorize block A2.

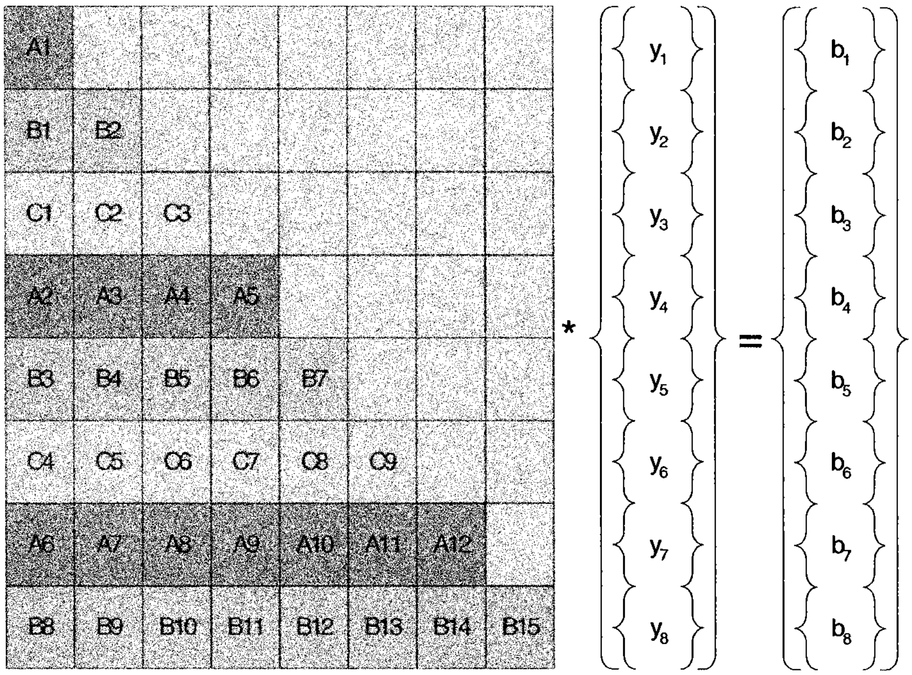
Now, processor 3 will be the master processor (i.e., the processor that will factorize the diagonal block), and the step mentioned above will be repeated for subsequent steps.

2.6.7 "Block" Forward Elimination Subroutine

From Section 2.6.1, the system of Eq. (2.11) can be rewritten as:

$$[U]^T \vec{y} = \vec{b}$$

or


A1
B1 B2
C1 C2 C3
A2 A3 A4 A5
B3 B4 B5 B6 B7
C4 C5 C6 C7 C8 C9
A6 A7 A8 A9 A10 A11 A12
B8 B9 B10 B11 B12 B13 B14 B15
*
y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8
=
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

In this forward substitution phase, three different types of processors are identified:

1. The master processor is the processor that calculates the "final" solution $\{x_i\}$ for this phase (i.e., processor 1 at starting point).
2. The next master processor is the processor that will calculate the final $\{x_i\}$ in the next loop of operation (i.e., processor 2 at starting point).
3. The workers are the processors that will calculate the updated value of $\{x_i\}$ in the next loop of operation (i.e., processor 3 at starting point).

The procedure from the given example starts at the forward substitution for the final solution $\{y_1\}$ by doing forward substitution of block A1 by processor 1, the master processor. Then, processor 1 will send the solution of $\{y_1\}$ to processor 2 (the next master processor) and 3 (the worker). After that, processor 1 will "partially" update {y4} and {y7} by doing forward of substitution blocks A2 and A6, respectively. For the next master processor, processor 2, it will update and calculate the final $\{y_2\}$ by doing forward substitution of blocks B1 and B2 respectively. After that, processor 2 will send the solution of $\{y_2\}$ to processor 3 (the next master processor) and 1 (the worker). At the same time, the worker, processor 3 will do forward substitution of block C1 to update $\{y_3\}$ and C4 to update $\{y_6\}$. The next step will be that processor 3 is the master processor, and the procedure will be repeated until all intermediate unknowns are solved.

2.6.8 "Block" Backward Elimination Subroutine

In this backward substitution phase, the unknown vector {x}, shown in Eq.(2.11), will be solved.

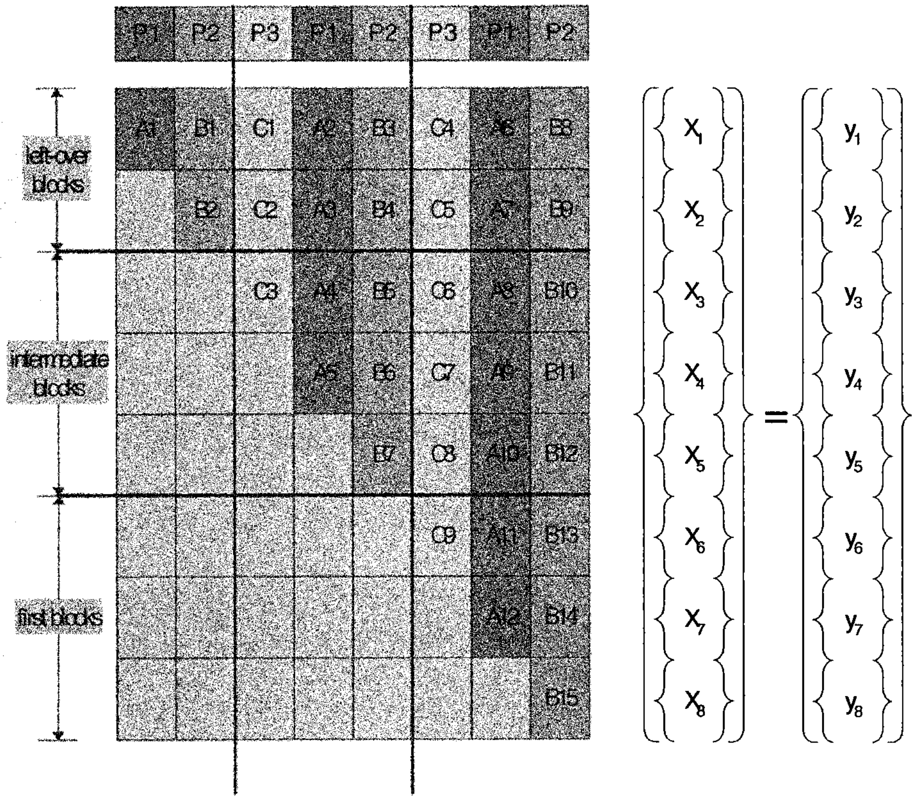


Figure 2.4 Backward Elimination Phase

From Figure 2.4, processor 2 will be the master of the starting process. The main ideas of the algorithm are to divide the upper-factorized matrix into a sub-group of block matrices, which have, normally ncb x np rows. Also, there are 3 types of sub-group blocks, which are the first blocks, the intermediate blocks, and the last blocks or leftover block. Then each processor will do backward substitution block by block from bottom to top and, if possible, from right to left. After doing backward substitution, each processor will send the solution (partially updated or final) to its left processor (i.e. processor 2 will send the solution to processor 1, processor 1 will send the solution to processor 3, and processor 3 will send the solution to processor 2). From the given example, processor 2 will do backward substitution of block B15 to find the final solution of X_8. Then, it will send X_8 to the adjacent (left) processor, processor 1, and start to do backward substitution of block B14 to find the partially updated solution of X_7. Then, processor 2 will send X_7 to processor 1. Now, processor 1 will do backward substitution of block A12 to find the final solution of X_7, while processor 2 is doing backward substitution of block B13 to find the partial updated solution of X_6. To clearly understand the steps involved in this phase, see Table 2.9.

Table 2.9 Processor Tasks in Backward Substitution Subroutine

Processor 1	Processor 2 (Master)	Processor 3	Remark
Idle (Waiting until B14 is finished)	B15	Idle (Waiting until A11 is finished)	Final solution $\{X_8\}$ is found
Idle	B14	Idle	$\{X_7\}_{\text{partial}}$
A12	B13	Idle	$\{X_7\}_{\text{final}}$, $\{X_6\}_{\text{partial}}$
A11	B12	Idle	$\{X_6\}_{\text{partial}}$, $\{X_5\}_{\text{partial}}$
A10	B11	C9	$\{X_6\}_{\text{final}}$, $\{X_5\}_{\text{partial}}$, $\{X_4\}_{\text{partial}}$
A9	B10	C8	$\{X_5\}_{\text{partial}}$, $\{X_4\}_{\text{partial}}$, $\{X_3\}_{\text{partial}}$
A8	B7	C7	$\{X_5\}_{\text{final}}$,

			$\{X_4\}_{partial}$, $\{X_3\}_{partial}$
Idle (Waiting until B6 is finished)	B6	C6	$\{X_4\}_{partial}$, $\{X_3\}_{partial}$
A5	B5	Idle (Waiting until A4 is finished)	$\{X_4\}_{final}$, $\{X_3\}_{partial}$
A4	B9	Idle	$\{X_3\}_{partial}$, $\{X_2\}_{partial}$
A7	B8	C3	$\{X_3\}_{final}$, $\{X_2\}_{partial}$, $\{X_1\}_{partial}$
A6	Idle (Waiting until C5 is finished)	C5	$\{X_2\}_{partial}$, $\{X_1\}_{partial}$
Idle (Waiting until B4 is finished)	B4	C4	$\{X_2\}_{partial}$, $\{X_1\}_{partial}$
A3	B3	Idle (Waiting until A3 is finished)	$\{X_2\}_{partial}$, $\{X_1\}_{partial}$
A2	Idle (Waiting until C2 is finished)	C2	$\{X_2\}_{partial}$, $\{X_1\}_{partial}$
Idle (Waiting until B1 is finished)	B2	C1	$\{X_2\}_{final}$, $\{X_1\}_{partial}$
Idle	B1	Done	$\{X_1\}_{partial}$
A1	Done	Done	$\{X_1\}_{final}$

2.6.9 "Block" Error Checking Subroutine

After the solution of the system of equations has been obtained, the next step that should be considered is error checking. The purpose of this phase is to evaluate how good the obtained solution is. There are four components that need to be considered:

1. X_{max} is the X_i that has the maximum absolute value (i.e., the maximum displacement, in structural engineering application).
2. Absolute summation of X_i

3. Absolute error norm of $[A] \cdot \vec{x} - \vec{b}$
4. Relative error norm is the ratio of absolute error norm and the norm of RHS vector $\vec{b}$.

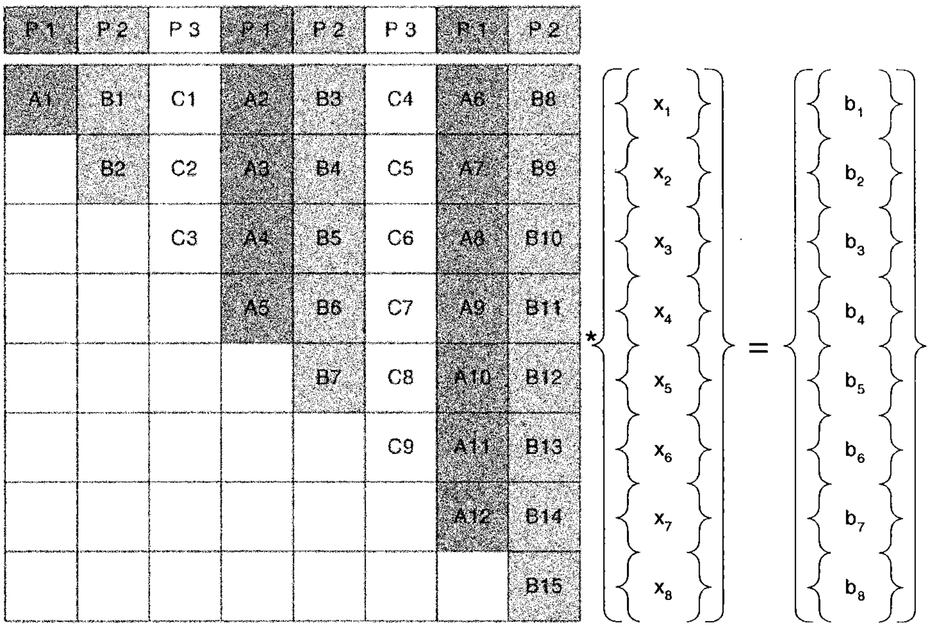


Figure 2.5 Error Norms Computation (Upper Triangular Portion)

The first two components can be found without any communication between each processor because the solution $\{x\}$ is stored in every processor. However, the next two components, absolute error norm and relative error norm, require communication between each processor because the stiffness matrix of the problem was divided in several parts and stored in each processor. Therefore, the parallel algorithm for error norm checking will be used to calculate these error norms. The key concepts of the algorithm are that every processor will calculate its own $[A] \cdot \vec{x} - \vec{b}$, and the result will be sent to the master processor, processor 1.

The procedure starts at each processor, which will calculate $[A]\{X\}_1$, which corresponds to $\{b_i\}$ for the lower part of the stiffness matrix. From Figure 2.6, processor 1 will partially calculate $[A]\{X\}_1$ by multiplying the lower part of [A1] with $\{X_1\}$.

Similarly, Processor 2 will partially calculate $[A]\{X\}_2$ by multiplying the lower part of [B1] and [B2] with $\{X_1\}$ and $\{X_2\}$, respectively. The step will be repeated until every processor finishes calculating the lower part of the stiffness matrix. In this step, there is no communication between processors.

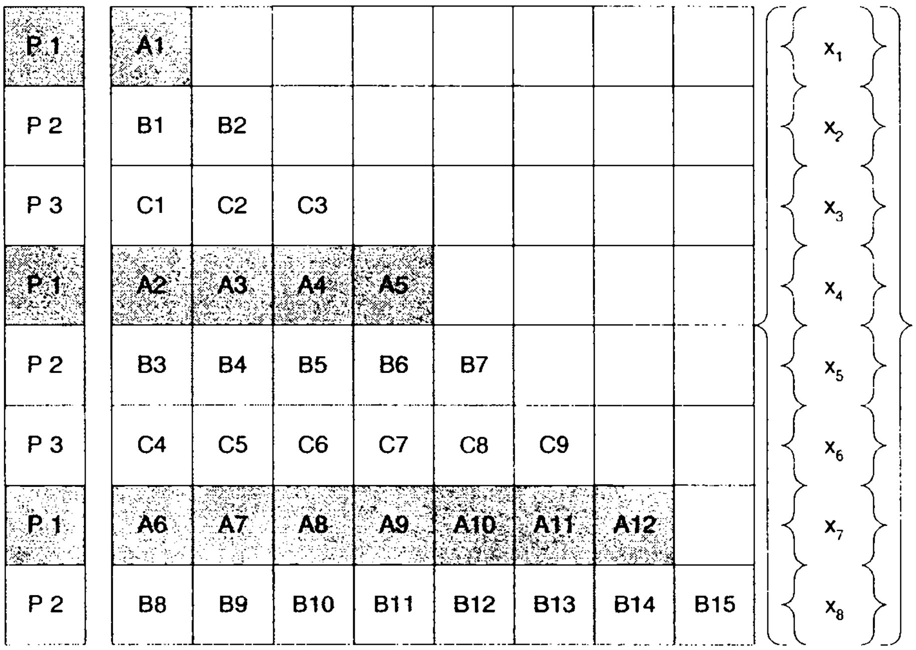


Figure 2.6 Error Norms Computation (Lower Triangular Portion)

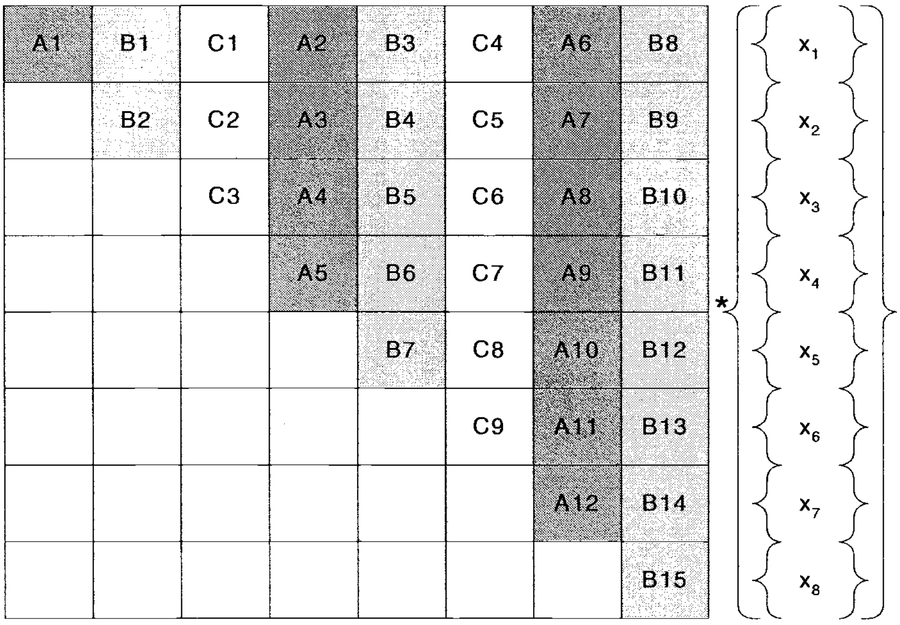


Figure 2.7 Error Norms Computation (Upper Triangular Portion)

Once the calculating for the lower part of the stiffness matrix is done, the next step will be the calculation of the diagonal and the upper part of the stiffness matrix. This procedure starts with the same concept as the calculation of the lower part. After the diagonal and the upper part is done, every processor will have its own $[A]\{X\}_i$.

Then, the worker processors will send $[A]\{X\}_i$ to the master processor. Therefore, processor 1 will have the complete values of $[A]\{X\}$. Next, processor 1 will find the difference between $[A]\{X\}$ and $\{b\}$, and the norm of this difference will be the absolute error norm. The absolute error norm will then be divided by the norm of the RHS vector to give the relative error norm.

2.6.10 Numerical Evaluation

Several full-size (completely dense) symmetrical matrix equations are solved, and the results (on different computer platforms) are compared and tabulated in Tables 2.10 – 2.12. All reported times are expressed in seconds. Performance of the "regular" versus "cache-based" matrix-matrix multiplication sub-routines is presented in Table 2.13. Results for a parallel SGI dense solver (on NASA LaRC SGI/Origin 2000, Amber Computer, 40 GB RAM), as compared to our developed software, are given in Tables 2.14 – 2.15.

It is interesting to note the "major difference" between our developed parallel dense solver (on the Sun 10k computer) as compared to the parallel SGI dense solver (on the NASA SGI/ORIGIN-2000 computer).

In particular, let's look at Tables 2.14 and 2.15, paying attention to the "time ratio" of "Forward-Backward / Factorization."

For 5,000 dense equations, using 8 (SGI Amber) processors, and with the near optimal block size ncb = 64, our developed parallel solver has the "time ratio" = (Forward & Backward / Factorization) = (3.5 / 19.1) = 18.32% as indicated in Table 2.15.

However, for the same problem size (= 5,000 equations), using 8 (NASA SGI/ORIGIN-2000, Amber) processors, the SGI Parallel Dense Solver library sub-routine has the "time ratio" = (Forward & Backward / Factorization) = (4.8 / 15.8) = 30.38%, as indicated in Table 2.14.

Thus, our developed software is much more efficient than the well-respected SGI software, especially for the Forward & Backward solution phases.

Finally, we show the results from porting the software to run on commodity computer systems. The software system using Windows 2000 workstation OS was developed using Compaq FORTRAN compiler (Version 6.1) and uses MPI-

Softtech's MPI library (Version 1.6.3). The same numerical experiments used with the two earlier hardware platforms yielded ncb = 32 and 64 showing somewhat similar performances. However, ncb = 64 was used as the block size. Moreover, the dense equations used in evaluating the performance came from actual finite element models. Table 2.16 shows the results for 4,888 equations while Table 2.17 is for 10,368 equations.

The performance of the commodity cluster is better than both the workstation platforms. When the matrices fit into memory (Table 2.16), the speedup is sub-linear. The speedup, however, is super-linear with larger problems and can be misleading.

This is because the matrices do not fit into memory and a larger number of page faults occur with the single processor version. Nevertheless, the overall throughput and performance is impressive.

Table 2.10 Two Thousand (2,000) Dense Equations (Sun 10k)

n = 2000	Factorization time				Forward Substitution				Backward Substitution			
	ncb=16	32	64	128	ncb=16	32	64	128	ncb=16	32	64	128
1 processor	16.243	9.896	8.552	8.337	0.176	0.173	0.173	0.175	0.110	0.097	0.098	0.097
2	7.637	5.170	4.873	5.108	0.087	0.040	0.066	0.261	0.236	0.083	0.048	0.039
4	3.736	3.028	3.010	3.548	0.028	0.034	0.044	0.194	0.109	0.046	0.028	0.026
8	2.634	2.076	2.470	3.186	0.025	0.029	0.047	0.210	0.057	0.027	0.020	0.021
16	1.766	1.569	1.723	✕	0.034	0.049	0.095	✕	0.038	0.020	0.016	✕

Table 2.11 Five Thousand (5,000) Dense Equations (Sun 10k)

n = 5000	Factorization time				Forward Substitution				Backward Substitution			
	ncb=16	32	64	128	ncb=16	32	64	128	ncb=16	32	64	128
1 processor	277.5	182.2	143.3	130.4	1.108	1.110	1.105	1.153	0.775	0.784	0.784	0.776
2	140.0	93.8	76.8	72.2	0.582	0.554	0.545	0.620	1.553	0.826	0.623	0.582
4	73.4	51.3	42.4	44.3	0.300	0.287	0.293	0.338	0.781	0.424	0.319	0.336
8	40.8	27.6	25.5	29.4	0.170	0.162	0.208	0.435	0.445	0.209	0.164	0.183
16	42.4	23.0	25.08	22.0	2.685	0.760	0.260	0.284	0.737	0.114	0.084	0.125

Table 2.12 Ten Thousand (10,000) Dense Equations (Sun 10k)

n = 10000	Factorization time				Forward Substitution				Backward Substitution			
	ncb=16	32	64	128	ncb=16	32	64	128	ncb=16	32	64	128
1 processor	2284.2	1548.6	1203.9	1059.2	4.565	4.881	4.541	4.548	3.317	3.560	3.323	3.302
2	1178.8	774.3	596.8	548.4	2.450	2.360	2.240	2.244	6.410	3.578	2.590	2.385
4	595.2	397.4	316.7	306.2	1.244	1.187	1.133	1.229	3.310	1.764	1.354	1.264
8	307.1	211.3	176.0	180.6	0.640	0.600	0.643	0.839	1.759	0.920	0.700	0.687
16	161.3	114.0	105.6	120.2	0.395	0.401	0.610	1.060	0.989	0.490	0.386	0.416

Table 2.13 Matrix-Matrix Multiplication Subroutines

	LIONS (Helios)			LIONS(Cancun)	NASA SGI/Origin-2000
	Regular version (seconds)	Cache version (seconds)	Cache Version (seconds)	Cache Version (seconds)	Cache Version (seconds)
1000x1000	177.405*	6.391*	6.420†	10.9†	9.5†
2000x2000	1588.03*	53.13*	54.02†	N/A	N/A

* using MPI_WTIME( )

† using etime( )

Table 2.14 Five Thousand (5,000) Dense Equations Timings Using SGI Parallel (SGI Amber) (seconds)

# of Processes	Factorization Time	Forward& Backward Substitution	Total Time	Speedup
1	87.4	4.2	91.6	1
2	48.8	4.6	53.4	1.7
4	23.8	4.4	28.2	3.2
8	15.8	4.8	20.6	4.4

Table 2.15 Five Thousand (5,000) Dense Equations Timings (SGI Amber) (seconds)

# of Processes	Factorization Time	Forward& Backward Substitution	Total Time	Speedup
1	103.1	3.4	106.5	1
2	55.9	2.8	58.7	1.8
4	31.8	1.7	33.5	3.2
8	19.1	3.5	22.6	4.7
10	16.2	4.4	20.6	5.2
16	12.9	5.4	18.3	5.8

Table 2.16 Five Thousand (4,888) Dense Equations Timings (Intel FEM) (seconds)

# of Processes	Factorization Time	Forward& Backward Substitution	Total Time	Speedup
1	61.1	0.2	61.3	1
2	33.7	0.7	34.4	1.8
3	24.2	0.5	24.7	2.5
4	19.3	0.5	19.8	3.1
5	16.6	0.4	17.0	3.6
6	14.6	0.3	14.9	4.1

Table 2.17 Ten Thousand (10,368) Dense Equations Timings (Intel FEM) (seconds)

# of Processes	Factorization Time	Forward& Backward Substitution	Total Time	Speedup
1	1258	1.1	1259.1	1
2	305	3.3	308.3	4.1
3	210	2.2	212.2	5.9
4	166	1.6	167.6	7.5
5	137	1.5	138.5	9.1
6	120	1.6	121.6	10.4
7	106	1.4	107.4	11.7

2.6.11 Conclusions

Detailed numerical algorithms and implementation for solving a large-scale system of dense equations have been described. Numerical results obtained from the developed MPI dense solver, and the SGI parallel dense solvers on different parallel computer platforms have been compared and documented.

The following conclusions, therefore, can be made:

1. The developed parallel dense solver is simple, efficient, flexible, and portable.
2. Using the developed parallel dense solver, a large-scale dense matrix can be stored across different processors. Hence, each processor only has to store "small portions" of the large dense matrix.
3. From Tables 2.10 – 2.12, it seems that when the problem size is getting bigger, the optimized number of columns per block that fits in a system cache will be approximately 64 columns per block.
4. Optimum performance of the developed parallel dense solver can be achieved by fine-tuning the block size on different computer platforms.

2.7 Developing/Debugging Parallel MPI Application Code on Your Own Laptop

A Brief Description of MPI/Pro

MPI/Pro is a commercial MPI middleware product. MPI/Pro optimizes the time for parallel processing applications.

MPI/Pro supports the full Interoperable Message Passing Interface (IMPI). IMPI allows the user to create heterogeneous clusters, which gives the user added flexibility while creating the cluster.

Verari Systems Software offers MPI/Pro on a wide variety of operating systems and interconnects, including Windows, Linux, and Mac OS X, as well as Gigabit Ethernet, Myrinet, and InfiniBand.

Web site

http://www.mpi-softtech.com/products/

Contact Information

Telephone:

Voice: (205) 397-3141
Toll Free: (866) 851-5244
Fax: (205) 397-3142

Sales support

sales@mpi-softtech.com

Technical support

mpipro-sup@mpi-softtech.com

Cost

- $100 per processor (8 processor minimum), one time payment
- Support and Maintenance are required for the first year at a rate of 20% per annum of the purchase price. Support ONLY includes: NT4SP5 and higher, 2000 Pro, XP Pro, and Windows 2003 Server.

Steps to Run Mpi/Pro on Windows OS

MPI/Pro requires Visual Studio 98 to run parallel FORTRAN code on Windows OS. Visual Studio 98 supports both FORTRAN and C programming languages. MPI is usually written in C programming language; therefore Visual Studio is sufficient for the user's needs.

A. Open Fortran Compiler Visual Studio

B. Open Project

Open a new "project" as a *Win 32 Console Application*. Give a name to your project, e.g., ***project1.*** This will create a folder named "project1."

C. Create a Fortran File

Create a new Fortran file under the project folder you have just created, e.g., *project1.for*, and when you create a file, click on the option "*Add to Project.*"

Then, type your program in this file. If you have other files or sub-routines to link to this file

- Create a file or files under the folder "*project1,*".
- Then, from the pulldown menu, go to Project/ Add to Project and select File.
- Select the files to be linked one by one and add them to your project.

D. Parallelizing Your Own Program With Mpi/Pro

There are a total of four call statements to type in a program running sequentially. But, first of all, one has to type the following line at the very beginning of the code:

include 'mpif.h'

Then, right after defining the data types and character strings, the following three call statements need to be typed:

call MPI_INIT (ierr)
call MPI_COMM_RANK (MPI_COMM_WORLD, me,ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, np,ierr)

where

me	= the ID of each processor *(e.g., me = 0, 1, 2, 3, 4, 5 if six processors are used)*
np	= total number of processors
ierr	= error message

In order to finalize the procedure of parallel programming, another call statement has to be placed at the end of the program:

call MPI_FINALIZE (ierr)

E. A Simple Example of Parallel MPI/FORTRAN Code

The entire list of the "trivial" MPI code is shown below:

```
implicit real*8(a-h,o-z)
include 'mpif.h'
real*8 a(16)

call MPI_INIT (ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, me, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)

if (me .eq. 0) then
open(5,file='rectdata.txt')
read(5,*) b,h
write(6,*) me,' b,h',b,h
endif

do i = 1,np
a(i) = (me+i)*2
enddo

write(6,*) 'me=',me,' a(-)',(a(i),i=1,np)
```

```
call sub1(me)
call sub2(me)

call MPI_FINALIZE(ierr)

stop
end
```

As you can see, there are two subroutine calls in the trivial MPI code. These sub-routines are added to the project as described in Section C. The sub-routines created for this example are listed below:

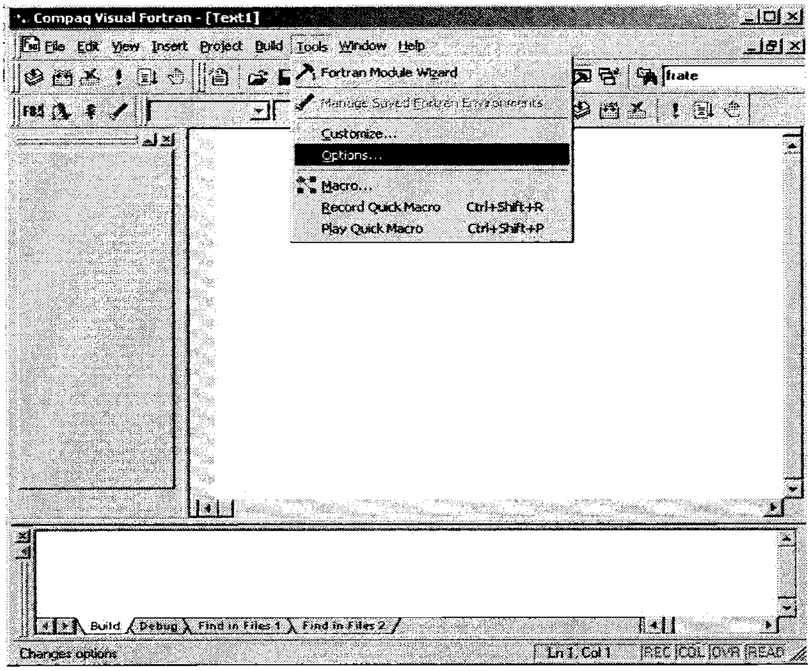
```
subroutine sub1(me)
write(6,*) me,' is in sub1 subroutine'
return
end

subroutine sub2(me)
write(6,*) me,' is in sub2 subroutine'
return
end
```

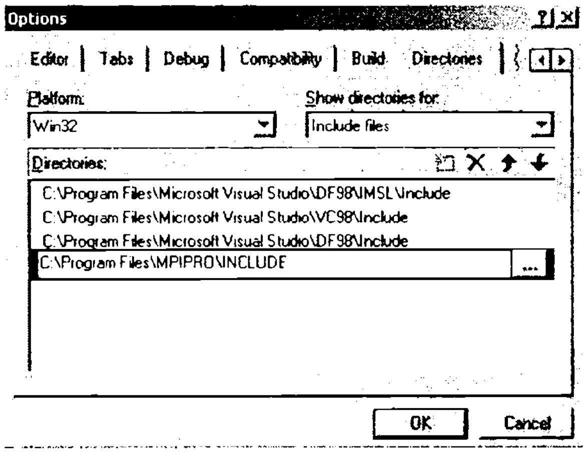
F. Before Compilation

A user makes some modifications in the FORTRAN compiler before he/she compiles his/her code. These modifications include defining the paths for the directory in which the file *'mpif.h'* is stored for the library files required by MPI/Pro.

1. From the pulldown menu, go to Tools then click on Options.


Compaq Visual Fortran - [Text1]
File Edit View Insert Project Build Tools Window Help
Fortran Module Wizard
Manage Saved Fortran Environments
Customize...
Options...
Macro...
Record Quick Macro Ctrl+Shift+R
Play Quick Macro Ctrl+Shift+P
Build Debug Find in Files 1 Find in Files 2
Changes options
Ln 1, Col 1 REC COL OVR READ

It will open up the Options window; click on Directories. Under the Show directories for sub-menu, the path for MPI/Pro include file is defined.



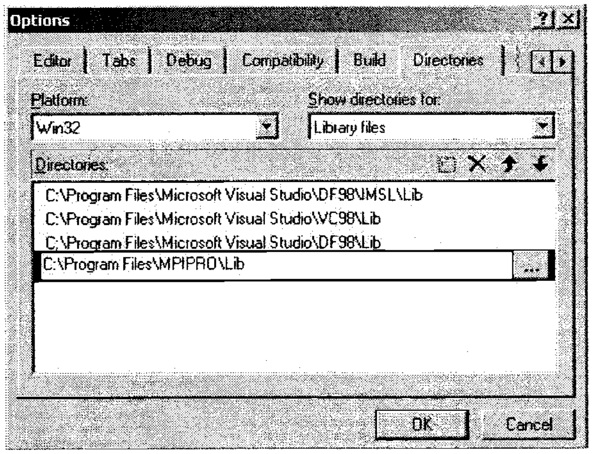
Platform: Win32

Show directories for: Include files

Directories:

C:\Program Files\Microsoft Visual Studio\DF98\IMSL\Include
C:\Program Files\Microsoft Visual Studio\VC98\Include
C:\Program Files\Microsoft Visual Studio\DF98\Include
C:\Program Files\MPIPRO\INCLUDE

The same procedure is repeated for MPI/Pro library files:



Platform: Win32

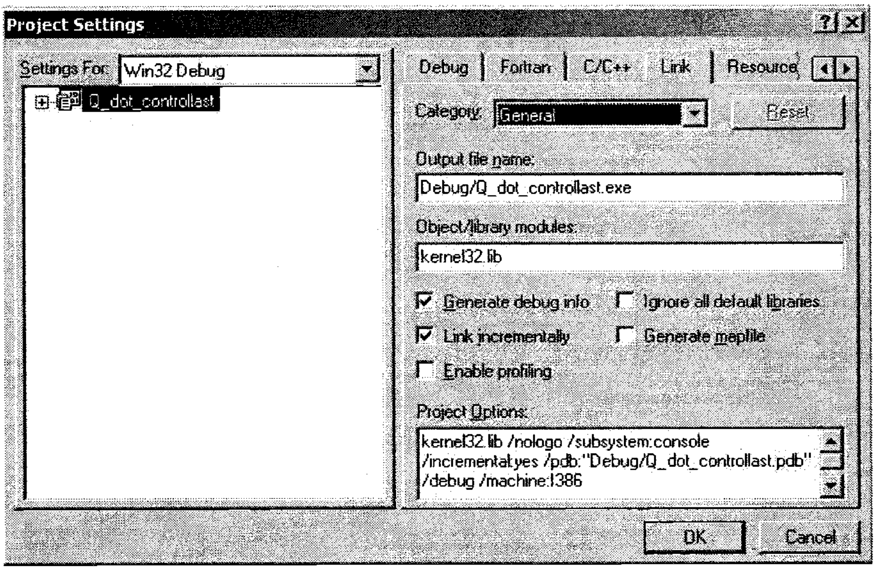
Show directories for: Library files

Directories:

C:\Program Files\Microsoft Visual Studio\DF98\IMSL\Lib
C:\Program Files\Microsoft Visual Studio\VC98\Lib
C:\Program Files\Microsoft Visual Studio\DF98\Lib
C:\Program Files\MPIPRO\Lib

2. From the pulldown menu, go to Project then click on Settings.

Here, choose Link and then General for Category.



Settings For: Win32 Debug

Q_dot_controllast

Category: General

Output file name: Debug/Q_dot_controllast.exe

Object/library modules: kernel32.lib

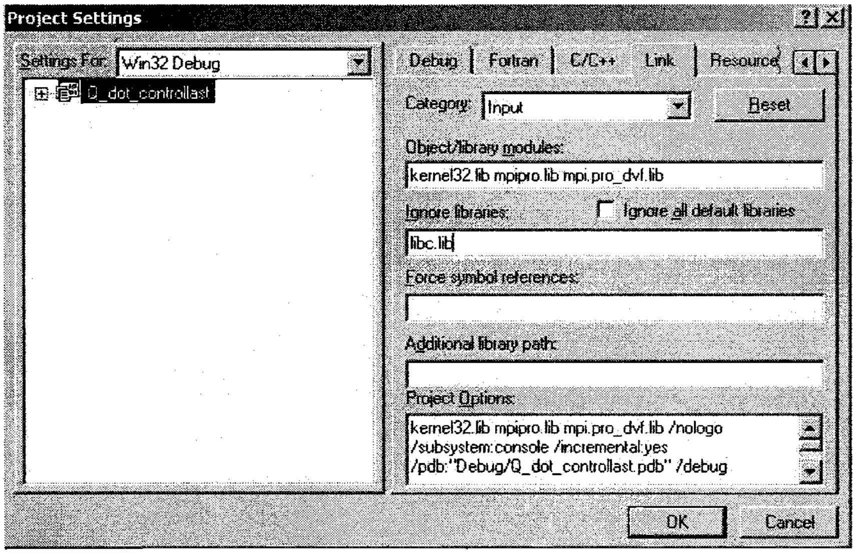
[✓] Generate debug info
[ ] Ignore all default libraries
[✓] Link incrementally
[ ] Generate mapfile
[ ] Enable profiling

Project Options:

kernel32.lib /nologo /subsystem:console /incrementalyes /pdb:"Debug/Q_dot_controllast.pdb" /debug /machine:I386

Add MPI/Pro libraries *mpipro.lib* and *mpipro_dvf.lib* to Object/libraries modules. Then, switch the Category to Input.

Type *libc.lib* in the Ignore libraries section as shown below:



Category: Input

Object/library modules:
kernel32.lib mpipro.lib mpi.pro_dvf.lib

Ignore libraries: ☐ Ignore all default libraries
libc.lib

Force symbol references:

Additional library path:

Project Options:

```
kernel32.lib mpipro.lib mpi.pro_dvf.lib /nologo
/subsystem:console /incremental:yes
/pdb:"Debug/Q_dot_controllast.pdb" /debug
```

G. Compilation

As the next step, the user is supposed to compile his/her program and create the executable file. To do this:

- From the pulldown menu, go to Build, then select Compile to compile the MPI code. When you do this, a folder named "Debug" will be created.
- Go to Build again, and select Build to create the executable of the MPI code. The executable file will be located under the folder "Debug."

H. Specifying the Computer Name(s)

In order to specify the machine name, create a file named *"Machines"* under the same directory where the executable file exists. Type in the computer name as your processor for parallel processing.

I. Running Parallel

1. Open up the window for a command prompt. To do this, go to Windows Start/Run and type "cmd" or "cmd.exe." This will run the command prompt.

2. In command prompt, go to the folder "Debug" where you have created the executable file.

3. Then type:

mpirun –np *number of processor* *executable file name*

J. Outputs of the Simple Demonstrated MPI/FORTRAN Code

Assuming six processors are used to "simulate" the parallel MPI environment on the "Laptop" personal computer, using the MPI code discussed in Section E. The results obtained from MPI/FORTRAN application code are shown below:

```
me=       1 a(-)  4.00000000000000      6.00000000000000
 8.00000000000000      10.0000000000000      12.0000000000000
 14.0000000000000
       1  is in sub1 subroutine
       1  is in sub2 subroutine

me=       4 a(-)  10.0000000000000      12.0000000000000
 14.0000000000000      16.0000000000000      18.0000000000000
 20.0000000000000
       4  is in sub1 subroutine
       4    is in sub2 subroutine

me=       2 a(-)  6.00000000000000      8.00000000000000
 10.0000000000000      12.0000000000000      14.0000000000000
 16.0000000000000
       2  is in sub1 subroutine
       2  is in sub2 subroutine

me=       3 a(-)  8.00000000000000      10.0000000000000
 12.0000000000000      14.0000000000000      16.0000000000000
 18.0000000000000
       3  is in sub1 subroutine
       3  is in sub2 subroutine

me=       5 a(-)  12.0000000000000      14.0000000000000
 16.0000000000000      18.0000000000000      20.0000000000000
 22.0000000000000
       5  is in sub1 subroutine
       5  is in sub2 subroutine
       0  b,h  6.00000000000000      8.00000000000000

me=       0 a(-)  2.00000000000000      4.00000000000000
 6.00000000000000      8.00000000000000      10.0000000000000
 12.0000000000000
       0  is in sub1 subroutine
       0  is in sub2 subroutine
```

2.8 Summary

In this chapter, a brief summary of how to write simple parallel MPI/FORTRAN codes has been presented and explained. Through two simple examples (given in Sections 2.2 – 2.3), it is rather obvious that writing parallel MPI/FORTRAN codes are essentially the same as in the sequential mode of FORTRAN 77, or FORTRAN 90, with some specially added parallel MPI/FORTRAN statements. These special MPI/FORTRAN statements are needed for the purposes of parallel computation and processors' communication (such as sending/receiving messages, summing /merging processor's results, etc.). Powerful unrolling techniques associated with "dot-product" and "saxpy" operations have also been explained. These unrolling strategies should be incorporated into any application codes to substantially improve the computational speed.

2.9 Exercises

2.1 Write a sequential FORTRAN (or C++) code to find the product $\{b\}=[A]*\{x\}$, using dot-product operations, where

$$[A]_{n\times n} = A_{i,j} = i + j$$

$$\{x\}_{n\times 1} = x_j = j$$

$$n=5000$$

2.2 Re-do problem 2.1 using saxpy operations.

2.3 Re-do problem 2.1 using unrolling level 10.

2.4 Re-do problem 2.1 using unrolling level 7.

2.5 Re-do problem 2.2 using unrolling level 10.

2.6 Re-do problem 2.1 using unrolling level 7.

2.7 Assuming 4 processors are available (NP = 4), write a parallel MPI/ FORTRAN (or C++) program to re-do problem 2.1.

2.8 Given a real array A(-) that contains "N" random values 0.00 and 1.00 (say, N = 10^6 numbers), and assuming there are "NP" processors available (say, NP = 4), write a MPI_parallel FORTRAN (or C^{++}) that will print the above "N" numbers in "increasing" order.