Chapter 32
MPI - Message Passing Interface

In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selections amongst them for the purposes of a calculating engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.
Ada Lovelace

Aim
The aims of this chapter is to provide a short introduction to MPI programming in Fortran.

32.1 Introduction

Documents for the MPI standard are available from the MPI Forum. Their web address is

```
http://www.mpi-forum.org
```

If you are going to do MPI programming we recommend getting hold of the document that refers to your implementation.

32.2 MPI Programming

MPI programming typically requires two components, a compiler and an MPI implementation. Two common ways of doing MPI programming are

- a cluster or multiple systems running MPI

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_32

- a single system running MPI

In both cases an MPI installation will normally provide an MPI daemon or service that can then be called from an MPI program.

32.3 Compiler and Implementation Combination

A number of commercial companies provide a combined bundle including

- Cray
- IBM
- Intel
- PGI

The Cray and IBM offerings will most likely be for a cluster. Intel and PGI provide products for both clusters and single systems. You should check their sites for up to date information.

32.4 Individual Implementation

A low cost option is to get hold of an MPI implementation that works with your existing compiler, and install it yourself on your own system.

The Intel MPI product is available as a free download for evaluation purposes.

There are a number of free MPI implementations, and details are given below for two of them.

32.4.1 MPICH2

They are based at Argonne National Laboratory

```
http://www.mpich.org/
```

MPICH2 is distributed as source (with an open-source, freely available license). It has been tested on several platforms, including Linux (on IA32 and x86-64), Mac OS/X (PowerPC and Intel), Solaris (32- and 64-bit), and Windows.

32.4.2 Open MPI

They can be found at

```
http://www.open-mpi.org/
```

They develop Open MPI on Linux, OS X, Solaris (both 32 and 64 on all platforms) and Windows (Windows XP, Windows HPC Server 2003/2008 and also Windows 7 RC).

32.5 Compiler and MPI Combinations Used in the Book

We have used a variety of compilers and MPI combinations, including

- Intel compiler + mpich2, Windows
- Intel compiler + Intel MPI, Windows
- gfortran + openmpi, openSuSe Linux
- Cray compiler, Hector Service
- Cray compiler, Archer Service
- PGI compiler, Hector Service
- IBM compiler, Met Office Slovakia

We haven't tried out all of the examples with all of the compiler and MPI implementations.

32.5.1 Cray Archer System

The Archer hardware consists of the Cray XC30 MPP supercomputer, external login nodes and postprocessing nodes, and the associated filesystems. There are 4920 compute nodes in Archer phase 2 and each compute node has two 12-core Intel Ivy Bridge Xeon series processors (2.7 GHz Intel E5-2697) giving a total of 118,080 processing cores. Each node has a total of 64 GB of memory with a subset of large memory nodes having 128 GB. A high-performance Lustre storage system is available to all compute nodes. There is no local disk on the compute nodes as they are housed in 4-node blades (the image below shows an XC30 blade with 4 compute nodes).

32.6 The MPI Memory Model

MPI is characterised generally by distributed memory and

- All threads/processes have access to their own private memory only
- Data transfer and most synchronization has to be programmed explicitly
- All data is private
- Data is shared explicitly by exchanging buffers in MPI terminology

but in this chapter we will also show the use of MPI on one system.

32.7 Example 1: Hello World

The first example is the classic hello world program.

```
program ch3201
  use mpi
  implicit none
  integer :: error_number
  integer :: this_process_number
  integer :: number_of_processes

  call mpi_init(error_number)
  call mpi_comm_size(mpi_comm_world, &
    number_of_processes, error_number)
  call mpi_comm_rank(mpi_comm_world, &
    this_process_number, error_number)
  print *, ' Hello from process ', &
    this_process_number, ' of ', &
    number_of_processes, 'processes!'
  call mpi_finalize(error_number)
end program ch3201
```

Let us look at each statement in turn.

```
use mpi
```

With most modern MPI implementations we can make available the MPI setup with a use statement. Older implementations required an include file option.

```
  call mpi_init( error_number )
```

This must be the first MPI routine called. The Fortran binding only takes one argument, an integer variable that is used to return an `error number`. It sets up the MPI environment.

```
call mpi_comm_size( mpi_comm_world, &
number_of_processes , error_number )
```

is typically the second MPI routine called. All MPI communication is associated with a so called communicator that describes the communication context and an associated set of processes. In this simple example we use the default communicator, called `mpi_comm_world`. The number of processes available is returned via the second argument. This means that the above program is duplicated on each process, i.e. `number_of_processes` determines how many copies are running.

```
call mpi_comm_rank( mpi_comm_world, &
  this_process_number , error_number )
```

The call above returns the process number for this process or copy of the program.

```
print *, " Hello from process " , &
  this_process_number , " of ", &
  number_of_processes , " processes!"
```

Each copy of the program will print out this message.

```
call mpi_finalize(error_number)
```

The call to `mpi_finalize` is the last call to the MPI system we need to make.

Here is the output from the Intel compiler and Intel MPI option under a Windows system.

```
mpiexec -n 8 ch3201
  Hello from process    0  of    8  processes!
  Hello from process    4  of    8  processes!
  Hello from process    1  of    8  processes!
  Hello from process    5  of    8  processes!
  Hello from process    7  of    8  processes!
  Hello from process    6  of    8  processes!
  Hello from process    3  of    8  processes!
  Hello from process    2  of    8  processes!
```

Notice that process numbering starts at 0. Note also that there is no particular order to the process numbers.

Here is the output from gfortran and openmpi on a openSuSe system. This is the same system as the above, as it is dual boot.

```
mpiexec -n 8 ch3201.out
  Hello from process    0  of    8  processes!
  Hello from process    1  of    8  processes!
  Hello from process    2  of    8  processes!
  Hello from process    3  of    8  processes!
  Hello from process    4  of    8  processes!
  Hello from process    5  of    8  processes!
  Hello from process    6  of    8  processes!
  Hello from process    7  of    8  processes!
```

Now the ordering is sequential.

Here is the output from the Cray Archer service. This uses 48 processes. The job is submitted as a batch job, via a queueing mechanism. This is a common mechanism on larger multi user systems.

```
  Hello world from image  16
  Hello world from image  6
  Hello world from image  13
  Hello world from image  25
  Hello world from image  34

lines deleted

  Hello world from image  38
  Hello world from image  44
  Hello world from image  35
  Hello world from image  28
  Hello world from image  33
  Hello world from image  32
  Hello world from image  30
  Hello world from image  29
```

The order appears to be pretty random!

32.8 Example 2: Hello World Using Send and Receive

The following is a variation of the above. In the first example we had no communication between processes. Sending and receiving of messages by processes is the basic MPI communication mechanism. The basic point-to-point communication

operations are send and receive. Their use is shown in the example below. These are blocking send and receive operations. A blocking send does not return until the message data and envelope have been safely stored away so that the sender is free to modify the send buffer. The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.

In this example process 0 is the master process and this communicates with every other process or program.

```
program ch3202
  use mpi
  implicit none
  integer :: error_number
  integer :: this_process_number
  integer :: number_of_processes
  integer :: i
  integer, dimension (mpi_status_size) :: status

  call mpi_init(error_number)
  call mpi_comm_size(mpi_comm_world, &
    number_of_processes, error_number)
  call mpi_comm_rank(mpi_comm_world, &
    this_process_number, error_number)
  if (this_process_number==0) then
    print *, ' Hello from process ', &
      this_process_number, ' of ', &
      number_of_processes, 'processes.'
    do i = 1, number_of_processes - 1
      call mpi_recv(this_process_number, 1, &
        mpi_integer, i, 1, mpi_comm_world, &
        status, error_number)
      print *, ' Hello from process ', &
        this_process_number, ' of ', &
        number_of_processes, 'processes.'
    end do
  else
    call mpi_send(this_process_number, 1, &
      mpi_integer, 0, 1, mpi_comm_world, &
      error_number)
  end if
  call mpi_finalize(error_number)
end program ch3202
```

The calls to

- `mpi_init`

- mpi_comm_size
- mpi_comm_rank
- mpi_finalize

are the same as in the first example. We have the additional code

- A test to see if we are process 0. If we are we then print out a message saying that we are process 0. We next loop from 1 to number_of_processes -1 and call mpi_recv.
- If we are not process 0 we make a call to mpi_send - remember that the program executes on all processes.

Let us look at the calls to mpi_recv and mpi_send in more depth. Here is an extract from the MPI 2.2 specification describing mpi_recv

```
<> buf(*)
  initial address of receive buffer
integer count
  number of elements in the receive buffer
datatype
  data type of each receive buffer element
source -   rank of source
tag -   message tag
comm -   communicator
status(mpi_status_size),
ierror
```

The following shows the mapping between MPI data types and Fortran data types.

```
mpi datatype        fortran datatype

mpi_integer            integer
mpi_real               real
mpi_double_precision   double precision
mpi_complex            complex
mpi_logical            logical
mpi_character          character(1)
```

our arguments to mpi_recv are

- this_process_number - process 0 is doing the receiving
- 1 item
- mpi_integer - an mpi_integer variable
- i - receive from this process
- 1 - tag
- mpi_comm_world - the communicator

- status - an integer array of size mpi_status_size
- error_number

Here is an extract from the 2.2 specification regarding mpi_send

```
<> buf(*) - initial address of send buffer
integer count - number of elements in send buffer
datatype - data type of each send buffer element
dest - rank of destination
tag - message tag
comm - communicator
ierror - error number\index{Error number}
```

the arguments to our mpi_send are

- this_process_number - send from this process
- 1
- mpi_integer
- 0 - send to this process number
- 1
- mpi_comm_world - the communicator
- error_number

and as you can see the sends and receives are in matching pairs.

Here is an Intel sample run.

```
Hello from process   0  of    8 processes.
Hello from process   1  of    8 processes.
Hello from process   2  of    8 processes.
Hello from process   3  of    8 processes.
Hello from process   4  of    8 processes.
Hello from process   5  of    8 processes.
Hello from process   6  of    8 processes.
Hello from process   7  of    8 processes.
```

Here is a Cray Archer sample run.

```
Hello from process  0  of  48 processes.
Hello from process  1  of  48 processes.
Hello from process  2  of  48 processes.
Hello from process  3  of  48 processes.
Hello from process  4  of  48 processes.

lines deleted
```

```
Hello from process  43  of  48 processes.
Hello from process  44  of  48 processes.
Hello from process  45  of  48 processes.
Hello from process  46  of  48 processes.
Hello from process  47  of  48 processes.
```

32.9 Example 3: Serial Solution for pi Calculation

We choose numerical integration in this example. The following integral

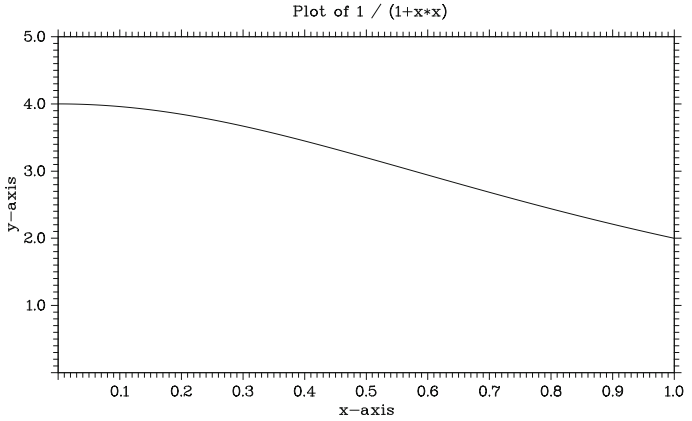
$$\int_0^1 \frac{4}{1+x^2}\,dx$$

is one way of calculating an approximation to π, and is a problem that is easy to parallelise. The integral can be approximated by

$$1/n \sum_1^n \frac{4}{1+\left(\frac{i-0.5}{n}\right)^2}$$

According to Wikipedia π to 50 digits is
3.14159265358979323846264338327950288419716939937510
Another way of calculating π is using the formula $4\,tan^{-1}(1)$ and in Fortran this is `4.0*atan(1.0)`.

Consider the following plot of the above equation.



To do the evaluation numerically we divide the interval between 0 and 1 into n sub intervals. The higher the value of n the more accurate our value of π will be, or should be.

Here is a serial program to do this calculation. The program is in three main parts. These are

- The module `precision_module` - to set the precision throughout the whole code.
- The module `timing_module` - a timing module to enable us to time parts of the program. We will be using this module throughout the parallel examples to provide information about the performance of the algorithms.
- the program - that actually does the integration.

The first two modules are straightforward and we will only cover the integration solution in depth. We will be using this integration example in this chapter on MPI and the subsequent two on OpenMP and coarray Fortran.

```
include 'precision_module.f90'

include 'timing_module.f90'

program ch3203
  use precision_module
  use timing_module
  implicit none
  integer :: i, j
  integer :: n_intervals
  real (dp) :: interval_width, x, total, pi
  real (dp) :: fortran_internal_pi

  call start_timing()
  n_intervals = 1000000
  fortran_internal_pi = 4.0_dp*atan(1.0_dp)
  print *, ' fortran_internal_pi = ', &
    fortran_internal_pi
  print *, ' '
  do j = 1, 4
    interval_width = 1.0_dp/n_intervals
    total = 0.0_dp
    do i = 1, n_intervals
      x = interval_width*(real(i,dp)-0.5_dp)
      total = total + f(x)
    end do
    pi = interval_width*total
    print 100, n_intervals, time_difference()
    print 110, pi, abs(pi-fortran_internal_pi)
    n_intervals = n_intervals*10
  end do
100 format (' N intervals = ', i12, ' time = ', &
    f8.3)
110 format (' pi = ', f20.16, /, &
```

```
      ' difference = ', f20.16)
    call end_timing()
    stop

  contains

    real (dp) function f(x)
      implicit none
      real (dp), intent (in) :: x

      f = 4.0_dp/(1.0_dp+x*x)
    end function f

  end program ch3203
```

The first part of the code has the declarations for the variables we will be using. These are

```
integer :: n_intervals
real (dp) :: interval_width, x, total, pi
real (dp) :: fortran_internal_pi
```

We have an integer variable for the number of intervals we will be using. We have made this of default integer type, which will be 32 bit on most platforms, and will be up to 2, 147, 483, 647.

We then have the following variables

- `interval_width`
- `z` - the variable we will be calculating numerically
- `total` - our total for the integration
- `pi` - our calculated value of π
- `fortran_internal_pi` - we use a common way of defining this using the internal `atan` function.

We then call the `start_timing` routine to print out details of the start time.

We next set the number of intervals. We choose 10 as an initial value. We will be doing the calculation for a number of interval sizes.

We calculate π using the `atan` intrinsic and print out its value. We will be using this value to determine the accuracy of our calculations.

We then have the loop that does the calculations for 9 values of the interval size from 10 to 1,000,000,000.

We calculate the interval width at the start of each loop and reset the total to zero at the start of each loop.

The following

```
do i = 1, n_intervals
  x = interval_width*(real(i,dp)-0.5_dp)
  total = total + f(x)
end do
```

is the code that actually does the integration. We calculate `x` each time round the loop and then use this calculated value in our call to our function, summing up as we go along. We need to subtract `a` as we need the mid point of the interval for our value of `x`.

The loop finishes and we then calculate the value of π and print out details of the number of intervals, the calculated value of `pi` and the difference between the internal value of π and the calculated value.

We also print out timing information about this calculation. We then increment the number of intervals and repeat the above.

We need to know how long the serial version takes and how accurate our calculated value for π is.

Here is output from this program on a couple of systems and compilers.

Compiler 1 - Intel compiler, Windows

```
2015/ 3/12 13:16:55 739
 fortran_internal_pi =    3.14159265358979

N intervals =      1000000 time =    0.000
pi =   3.1415926535899033
difference =   0.0000000000001101
N intervals =     10000000 time =    0.031
pi =   3.1415926535896861
difference =   0.0000000000001070
N intervals =    100000000 time =    0.281
pi =   3.1415926535902168
difference =   0.0000000000004237
N intervals =   1000000000 time =    2.871
pi =   3.1415926535897682
difference =   0.0000000000000249
2015/ 3/12 13:16:58 922
```

Compiler 2 - gfortran, Windows

```
2015/ 3/12 15:14:42 110
 fortran_internal_pi =    3.1415926535897931

N intervals =      1000000 time =    0.016
pi =   3.1415926535899601
difference =   0.0000000000001670
```

```
 N intervals =     10000000 time =    0.016
 pi =   3.1415926535897216
 difference =   0.0000000000000715
 N intervals =    100000000 time =    0.281
 pi =   3.1415926535900236
 difference =   0.0000000000002305
 N intervals =   1000000000 time =    2.793
 pi =   3.1415926535896523
 difference =   0.0000000000001408
 2015/ 3/12 15:14:45 214
```

Compiler 3 - Cray, Archer Service. Hardware details of this system are given earlier.

```
sttp1553@eslogin008:~> ./ch3003.x
 2015/ 3/22 11:42: 5  50
  fortran_internal_pi =  3.1415926535897931

 N intervals =      1000000 time =    0.000
 pi =   3.1415926535899033
 difference =   0.0000000000001101
 N intervals =     10000000 time =    0.023
 pi =   3.1415926535896861
 difference =   0.0000000000001070
 N intervals =    100000000 time =    0.207
 pi =   3.1415926535902168
 difference =   0.0000000000004237
 N intervals =   1000000000 time =    2.074
 pi =   3.1415926535897682
 difference =   0.0000000000000249
 2015/ 3/22 11:42: 7 356
 STOP
```

The three sample serial runs provide us with information that we can use as a basis for an analysis of our parallel solution. We have information about the accuracy of the solution and timing details.

32.10 Example 4: Parallel Solution for pi Calculation

This example is a parallel solution to the above problem using MPI. We only show the parallel program. The precision and timing modules are the same as in the previous example.

```
include 'precision_module.f90'

include 'timing_module.f90'

program ch3204
  use precision_module
  use timing_module
  use mpi
  implicit none
  real (dp) :: fortran_internal_pi
  real (dp) :: partial_pi
  real (dp) :: total_pi
  real (dp) :: width
  real (dp) :: partial_sum
  real (dp) :: x
  integer :: n
  integer :: this_process
  integer :: n_processes
  integer :: i
  integer :: j
  integer :: error_number

  call mpi_init(error_number)
  call mpi_comm_size(mpi_comm_world, &
    n_processes, error_number)
  call mpi_comm_rank(mpi_comm_world, &
    this_process, error_number)
  n = 100000
  fortran_internal_pi = 4.0_dp*atan(1.0_dp)
  if (this_process==0) then
    call start_timing()
    print *, ' fortran_internal_pi = ', &
      fortran_internal_pi
  end if
  do j = 1, 5
    width = 1.0_dp/n
    partial_sum = 0.0_dp
    do i = this_process + 1, n, n_processes
      x = width*(real(i,dp)-0.5_dp)
      partial_sum = partial_sum + f(x)
    end do
    partial_pi = width*partial_sum
    call mpi_reduce(partial_pi, total_pi, 1, &
      mpi_double_precision, mpi_sum, 0, &
      mpi_comm_world, error_number)
```

```
    if (this_process==0) then
      print 100, n, time_difference()
      print 110, total_pi, abs(total_pi- &
        fortran_internal_pi)
    end if
    n = n*10
  end do
  call mpi_finalize(error_number)
100 format (' N intervals = ', i12, ' time = ', &
    f8.3)
110 format (' pi = ', f20.16, /, &
    ' difference = ', f20.16)

contains

  real (dp) function f(x)
    implicit none
    real (dp), intent (in) :: x

    f = 4.0_dp/(1.0_dp+x*x)
  end function f

end program ch3204
```

The first difference is the

```
use mpi
```

statement. This makes available the MPI functionality. We next have several variable declarations.

```
real (dp) :: fortran_internal_pi
real (dp) :: partial_pi
real (dp) :: total_pi
real (dp) :: width
real (dp) :: partial_sum
real (dp) :: x
integer :: n
integer :: this_process
integer :: n_processes
integer :: i
integer :: j
integer :: error_number
```

The variables `partial_pi`, `total_pi` and `partial_sum` are required by our parallel algorithm. The variable n is the number of intervals and we start this at 100,000 rather than 10 as we have seen from the serial solution that there are quite large differences between the internal value of pi and the calculated value below 100,000.

The variables `this_process`, `n_processes` and `error_number` are required for the MPI solution.

The real work is done in the following do loop.

```
do i = this_process + 1, n, n_processes
  x = width*(real(i,dp)-0.5_dp)
  partial_sum = partial_sum + f(x)
end do
```

The key is to split up the work of the calculation between the processes we have available. The following shows how the work will be split up for n = 10 and with the number of processes ranging from 1 to 8.

```
n_processes=1 do i=1,n,1  1,2,3,4,5,6,7,8,9,10
n_processes=2 do i=1,n,2  1,3,5,7,9
              do i=2,n,2  2,4,6,8,10
n_processes=4 do i=1,n,4  1,5,9
              do i=2,n,4  2,6,10
              do i=3,n,5  3,7
              do i=4,n,4  4,8
n_processes=8 do i=1,n,8  1,9
              do i=2,n,8  2,10
              do i=3,n,8  3
              do i=4,n,8  4
              do i=5,n,8  5
              do i=6,n,8  6
              do i=7,n,8  7
              do i=8,n,8  8
```

The above also shows how the algorithm balances the load of the computation across the processes.

Each process has its own `partial_sum` and `partial_pi`. We then use the call to the MPI subroutine `mpi_reduce` to calculate the total value of pi from the partial values of pi. Here is the MPI description of the `mpi_reduce` routine

```
MPI_REDUCE( sendbuf, recvbuf, count,
            datatype, op, root, comm)
IN  sendbuf address of send buffer (choice)
OUT recvbuf address of receive buffer
```

```
    (choice, significant only at root)
IN  count number of elements in send buffer
    (non-negative integer)
IN  datatype data type of elements of send buffer
    (handle)
IN  op reduce operation (handle)
IN  root rank of root process (integer)
IN  comm communicator (handle)
```

and

```
partial_pi is our send buffer
total_pi is our receive buffer
1 - the number of elements
mpi_double_precision - the type of the elements
mpi_sum - the reduction operation
0 - the root process
mpi_comm_world  - the communicator
error_number - the error number
```

We then control the printing from process 0.
Here is sample output from the Intel compiler on a 6 core AMD system.

```
mpiexec -n 6 ch3004.exe
 2015/ 3/12 13:16:39 671
  fortran_internal_pi =    3.14159265358979
 N intervals =       100000 time =    0.000
 pi =   3.1415926535981256
 difference =   0.0000000000083324
 N intervals =      1000000 time =    0.000
 pi =   3.1415926535898762
 difference =   0.0000000000000830
 N intervals =     10000000 time =    0.000
 pi =   3.1415926535897674
 difference =   0.0000000000000258
 N intervals =    100000000 time =    0.062
 pi =   3.1415926535897389
 difference =   0.0000000000000542
 N intervals =   1000000000 time =    0.637
 pi =   3.1415926535898402
 difference =   0.0000000000000471
```

We get a nearly linear speed up over the serial version, which shows how good the parallel solution is. Note that the time value is not the total time taken by all

processes, but rather the effective running time of the program. If we are sat in front of the pc the program would complete in about a quarter of the time of the serial version. The numerical results are similar to the serial solution.

Table 32.1 summarises the output from the Intel compiler on an Intel I7 system. The table has the execution time details when running the program on 1 to 8 cores. The timing for cores 1–4 are for the program runs on real physical cores. The timing for cores 5–8 are when running on hyperthreaded cores. The execution time is worse when running on 5–7 cores. You should time your programs on hyperthreaded systems to see if running on the extra cores brings any benefit.

Table 32.1 Intel I7 with hyperthreading

	Cores							
Intervals	1	2	3	4	5	6	7	8
100,000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
1,000,000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
10,000,000	0.016	0.016	0.012	0.000	0.016	0.000	0.000	0.016
100,000,000	0.234	0.109	0.078	0.062	0.094	0.094	0.078	0.062
1,000,000,000	2.203	1.141	0.816	0.609	0.984	0.812	0.703	0.594

As can be seen the performance for 5–8 cores is similar to that for 4 cores. Cores 5–8 represent the hyperthreaded cores.

Here is the output from the Cray at the Archer service. This is for 48 processes running on 2 nodes.

```
2015/ 3/21  1:11:47 841
 fortran_internal_pi =  3.1415926535897931
N intervals =      1000000 time =    0.004
pi =   3.1415926535898757
difference =   0.0000000000000826
N intervals =     10000000 time =    0.000
pi =   3.1415926535897958
difference =   0.0000000000000027
N intervals =    100000000 time =    0.006
pi =   3.1415926535897909
difference =   0.0000000000000022
N intervals =   1000000000 time =    0.054
pi =   3.1415926535897949
difference =   0.0000000000000018
```

32.11 Example 5: Work Sharing Between Processes

This example looks at one way of splitting work up between processes. We use the process number of determine which process does which work.

```
program ch3205
  use mpi
  implicit none
  integer :: error_number
  integer :: this_process_number
  integer :: number_of_processes
  integer, dimension (mpi_status_size) :: status
  integer, allocatable, dimension (:) :: x
  integer :: n
  integer, parameter :: factor = 5
  integer :: i, j, k
  integer :: start
  integer :: end
  integer :: recv_start

  call mpi_init(error_number)
  call mpi_comm_size(mpi_comm_world, &
    number_of_processes, error_number)
  call mpi_comm_rank(mpi_comm_world, &
    this_process_number, error_number)
  n = number_of_processes*factor
  allocate (x(1:n))
  x = 0
  start = (factor*this_process_number) + 1
  end = factor*(this_process_number+1)
  print 100, this_process_number, start, end
  do i = start, end
    x(i) = i*factor
  end do
  do i = 1, n
    print 110, this_process_number, i, x(i)
  end do
  if (this_process_number==0) then
    do i = 1, number_of_processes - 1
      recv_start = (factor*i) + 1
      call mpi_recv(x(recv_start), factor, &
        mpi_integer, i, 1, mpi_comm_world, &
        status, error_number)
    end do
```

```
    else
      call mpi_send(x(start), factor, mpi_integer, &
        0, 1, mpi_comm_world, error_number)
    end if
    if (this_process_number==0) then
      do i = 1, n
        print 120, i, factor, x(i)
      end do
    end if
    call mpi_finalize(error_number)
100 format (' Process number = ', i3, ' start ', &
    i3, ' end ', i3)
110 format (1x, i4, ' i ', i4, ' x(i) ', i4)
120 format (1x, i4, ' * ', i2, ' = ', i5)
end program ch3205
```

What we are going to do is allocate an array based on the number of processes and then split the (simple) work on the array up between the processes. We will calculate array indices from the process numbers.

```
n = number_of_processes*factor
```

This statement calculates the array size based on the number of processes and a constant factor.

```
allocate (x(1:n))
```

This statement allocates the array.

```
x = 0
```

This statement initialises the whole array to zero. The following statements define the start and end points for the array processing for each process.

```
start = (factor*this_process_number) + 1
end = factor*(this_process_number+1)
```

and partition the work up between the processes. Each process will have its own start and end values. The following do loop does the work:

```
do i = start, end
  x(i) = i*factor
end do
```

and all we are doing as this is filling sections of the array up with data based in process numbers.

The following

```
if (this_process_number==0) then
  do i = 1, number_of_processes - 1
    recv_start = (factor*I) + 1
    call mpi_recv(x(recv_start), &
      factor,mpi_integer,i,1,mpi_comm_world,&
      status ,error_number)
  end do
else
  call mpi_send(x(start),factor, &
    mpi_integer,0,1,mpi_comm_world,error_number)
end if
```

uses sends and receives to transfer the updated array sections back to process zero. We are using `recv_start` to specify the starting point for the array transfer, and `x(start)` is the starting point for the transfer from the x array to process zero.

Here is sample output from the program when the number of processes is three.

```
mpiexec -n 3 ch3205
 Process number =   2 start  11 end  15
 Process number =   1 start   6 end  10
    1 I    1 x(i)    0
    1 I    2 x(i)    0
    1 I    3 x(i)    0
    1 I    4 x(i)    0
    1 I    5 x(i)    0
    1 I    6 x(i)   30
    1 I    7 x(i)   35
    1 I    8 x(i)   40
    1 I    9 x(i)   45
    1 I   10 x(i)   50
    1 I   11 x(i)    0
    1 I   12 x(i)    0
 Process number =   0 start   1 end   5
    0 I    1 x(i)    5
    0 I    2 x(i)   10
    0 I    3 x(i)   15
    0 I    4 x(i)   20
    0 I    5 x(i)   25
    0 I    6 x(i)    0
    0 I    7 x(i)    0
```

```
 0 I    8 x(i)    0
 0 I    9 x(i)    0
 0 I   10 x(i)    0
 0 I   11 x(i)    0
 0 I   12 x(i)    0
 0 I   13 x(i)    0
 0 I   14 x(i)    0
 0 I   15 x(i)    0
 1 *  5 =     5
 2 *  5 =    10
 3 *  5 =    15
 4 *  5 =    20
 5 *  5 =    25
 6 *  5 =    30
 7 *  5 =    35
 2 I    1 x(i)    0
 2 I    2 x(i)    0
 2 I    3 x(i)    0
 2 I    4 x(i)    0
 2 I    5 x(i)    0
 2 I    6 x(i)    0
 2 I    7 x(i)    0
 2 I    8 x(i)    0
 2 I    9 x(i)    0
 2 I   10 x(i)    0
 2 I   11 x(i)   55
 2 I   12 x(i)   60
 2 I   13 x(i)   65
 2 I   14 x(i)   70
 2 I   15 x(i)   75
 1 I   13 x(i)    0
 1 I   14 x(i)    0
 1 I   15 x(i)    0
 8 *  5 =    40
 9 *  5 =    45
10 *  5 =    50
11 *  5 =    55
12 *  5 =    60
13 *  5 =    65
14 *  5 =    70
15 *  5 =    75
```

So with three processes we have an array of size 15, and the work that each process does is

```
Process number =   0 start   1 end   5
Process number =   1 start   6 end  10
Process number =   2 start  11 end  15
```

and each process works on its own section of the array. At the end we use the sends and receives to make sure that the x array on process zero now has all of the updated values.

This code achieves load balancing across the processes.

32.12 Summary

The programs in this chapter provide an introduction to the use of MPI to achieve parallel programs in Fortran. We have also seen some of the timing benefits of parallel programming with MPI.

32.13 Problem

32.1 Compile and run the programs with your compiler and implementation of MPI. You should get similar results.