Outline:

> Why MPI? - Simple MPI.
> Basic Considerations in Message Passing & How to Combine Messages
> Collective Communication

Analyze timings.

Timings
Topologies
Nonblocking and Persistent Communication

fortran files displayed:
- monte.f
- ring.f
- ring2.f
- monte3.f
- monte2.f
- nonblock.f
- persist3.f
Why Parallel

Science and engineering computations are now typically made on computers. Computers enable realistic modeling of physical phenomenon, allowing us to examine more than just special cases. e.g., "dusty deck" Fortran codes may represent a core engineering expertise. To improve validity of physical models, we typically add more grid points. For example, we might want a 1K by 1K by 1K grid. If there is one eight byte floating point number for each element of the grid, that would be 8 Gbytes.

One reason to go to parallel computation is just to have sufficient RAM available. Many of the standard finite element packages still are not well parallelized, but nonetheless require lots of memory to model in sufficient detail. The fastest computers now (the ones that solve physical problems with the best resolution) are parallel computers.

The dominant style of parallel computer is an MIMD (Multiple Instruction Multiple Device) computer. The style of programming I'll talk about here is Single Instruction Multiple Device (SIMD). This uses "if this group of nodes, else that group of nodes") constructs so that SENDS, RECEIVES, and synchronization points can all appear in the same code.
If all processors can access the same RAM, the computer is said to be shared memory.

If memory is distributed, (a distributed message passing is the usual style of programming.

**Why MPI? -- Simple MPI.**

MPI (Message Passing Interface) is a standard. Before MPI (and PVM) it was necessary to rewrite the message passing parts of routines for every new parallel platform that came along.

Considering the great number of "dead" architectures, this meant that by the time code worked well (a year or two) the machine it was written for was almost out of date.

Connection Machines, DAP, BBN, Kendall Square, DEC, Intel Message Passing, Sequent, Cray, SGI ...

MPI programs work on both shared memory and distributed memory
machines. So just as we have "dusty deck" Fortran codes, we will have legacy MPI-FORTRAN and MPI-C codes. Hopefully, they will be maintainable? Portably fast?

MPI is a very rich (complicated) library. But it is not necessary to use all the features. An advantage to using a simple subset is these have been optimized by most vendors. Also it makes thinking about programming easy. On the other hand, it is fun to look up other functions and see if they will simplify programming. Often the problem you are having is one addressed by some MPI construct.

There are two sets of sample programs. Due mainly to Peter Pacheco from the University of San Francisco. ppmpi_c and ppmpi_f in C and Fortran respectively. Download from anonymous ftp.

ftp cs.fit.edu
login: anonymous
cd pub/howell
get hpc.tar
get mpi_rweed.ppt -- is a power point presentation migrated from Mississippi State (Dr. Rick Weed).
get pachec.tar -- for a more complete set of example MPI codes.

References
Parallel Programming with MPI -- Peter Pacheco – Morgan Kaufman Press. A good introduction to parallel programming – and to MPI, examples in C.


RS/6000 SP: Practical MPI Programming—www.redbooks.ibm.com by Yukiya Aoyama and Jun Nakano

www-unix.mcs.anl.gov/mpi/standard.html has the actual MPI standard documents
You can look at the mpi.h, fmpi.h, mpio.h file on your system. Usually these are found in /usr/include

I'm presenting codes in Fortran.
(Argue: Fortran 77 is the subset of C which are most useful for scientific computing, other features of C, e.g. pointer, arithmetic are likely to slow performance).

Fortran calling arguments to MPI routines are different than C. Usually, an ierr argument is appended. Where a C function would return an integer zero on successful return, the Fortran subroutine returns ierr as zero instead.

**SAMPLE EASY CODES**

Monte Carlo codes run many instances of a given event. The original Monte Carlo calculations were run to model an H-Bomb. The Russians already had one, how by hook or by crook to model the interactions of a
neutron?

Monte Carlo calculations of a given constant (e.g., mean free path of a neutron or area under a curve) have

\[ \text{Error} = O\left(\frac{1}{\sqrt{\text{number of simulations}}} \right) \]

So to get one more digit of accuracy we have to multiply the number of simulations by one hundred. Hence a need for many simulations, i.e., so a need for parallel computation.

The simulations are independent and require little communication, so Monte Carlo codes are known as “embarrassingly parallel”.

**FILE: monte.f**

c
 c Template for a Monte Carlo code.
 c
 c The root processor comes up with a list of seeds
 c which it ships to all processors.
 c
 c In a real application, each processor would compute
 c something and send it back. Here they compute
a vector of random numbers and send it back.

This version uses a loop of sends to mail out the seeds. And uses a loop to send data back to root.

program monte

include 'mpif.h'

call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)

c function
    integer string_len

    call MPI_Init(ierr)

    call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
if (my_rank.eq.0) then
  print*, 'input random seed'
  read*, iseed
endif

c------------------------------------------------------------------------
c if the timing results seem peculiar, try uncommenting the next line
c   call MPI_BARRIER(MPI_COMM_WORLD,ierr) !
   !
   !  What difference is there?
startim = MPI_Wtime()
if (my_rank.eq.0) then
  initseed = int(random()*1000)
  startim = MPI_Wtime()
do dest = 1, p-1    ! CPU 0 loops through p-1 sends
  initseed = int(random()*1000)
tag = 0

call MPI_Send(initseed, 1, MPI_INTEGER, +
              dest, tag, MPI_COMM_WORLD, ierr)
c------------------------------------------------------------------------
c Send message consisting of
  initseed  -- arg 1, message sent
  1          -- arg 2, length of message
  MPI_INTEGER  -- arg 3, type of data sent
  dest       -- arg 4, rank of processor to which message sent
  tag        -- arg 5, some integer, needs to be matched by RECV
  MPI_COMM_WORLD  -- arg 6, handle of communicator, matched by RECV
  ierr       -- arg 7, output from MPI_SEND, will be 0 if successful
This call is blocking. Code will not proceed until the receiving processor signals that it has started to receive.

end do

else ! each of p-1 CPUs gets a message from CPU 0
   root = 0
   tag = 0
   call MPI_Recv(initseed, 100, MPI_CHARACTER, root,
                   tag, MPI_COMM_WORLD, status, ierr)

Receive message consisting of
initseed  -- arg 1 message sent
100      -- arg 2, maximal length of message
MPI_INTEGER -- arg 3, type of data sent
root      -- arg 4, rank of processor which sent message
            (could use a wild card)
tag       -- arg 5, some integer, needs to be matched by SEND
            (could use a wild card)
MPI_COMM_WORLD -- arg 6, handle of communicator, matched by SEND
            (no wild card allowed)
status    -- arg 7 integer array status(MPI_STATUS_SIZE)
ierr      -- arg 8, output from MPI_SEND, will be 0 if successful
The receive is blocking. Code will not go to next step until the receive is completed.

call MPI_Get_count(status, MPI_INTEGER, size, ierr)

This call tells us how long the passed message came out to be
Information about the received message is found in status vector
c ---------------------------------------------------------------------------------------------------------
    endif  ! input phase done

c-----------------------------------------------------------------------
c  Left out -- a body of code that does a bunch of particle tracking
c           stuff to produce the double precision vector ans
c-----------------------------------------------------------------------

   do i=1,10
       ans(i) = rand()  ! at least we initialize stuff to send back.
   end do

   if (my_rank.eq.0) then
       tag = 1
       do source = 1,p-1
           call MPI_RECV(ans2, 10, MPI_DOUBLE_PRECISION,
                           source, tag, MPI_COMM_WORLD, status, ierr)
           do i=1,10
               ans(i) = ans(i) + ans2(i)
           end do
       end do
   else
       tag = 1
       call MPI_SEND(ans, 10, MPI_DOUBLE_PRECISION, root,
                       tag, MPI_COMM_WORLD, ierr)
   endif
   if(my_rank.eq.0) then
       c             do some stuff to process and output ans
   endif
   entim =  MPI_Wtime() - startim
call MPI_Finalize(ierr)
print*, 'elapsed time =', entim, '      my_rank=', my_rank
end

Let's look a bit at the arguments for the send and receive.

MPI_SEND,
1st argument, variable to be sent
2nd argument, integer number of items in variable, here 1
3rd , data type--different Fortran and C bindings !!
4th , dest -- integer rank of processor to which message is sent
5th , tag -- integer message tag
6th , handle for MPI communicator
7th , ierr , integer for error message, this argument not used in C ,
       output

MPI_RECV
1st arg, variable to be received (output)
2nd arg, integer of number of variables in message , This is
       an upper bound for the receive
3rd arg, data type
4th arg, source -- integer rank of processor sending message
(or could be MPI_ANY_SOURCE)
5th arg, tag – same integer as matching send,
(or could be MPI_ANY_TAG)
6th arg handle for MPI communicator, must match sending communicator
7th arg status, integer status(MPI_STATUS_SIZE) – output
8th arg ierr, output, 0 for successful return

The integer status(MPI_SOURCE) tells us the processor rank of the source of the received message. The integer status(MPI_TAG) sends us the tag off the received message. There's also a value status (MPI_ERROR).

In general, I won't write out the MPI arguments in such detail, but as with any C or Fortran library, keeping good track of subroutine arguments is a first key to successfully using a call.

It is often helpful to write a small program to illustrate and verify the action of the routine. On a given architecture, you need to know the correct data
types to match the integers used in MPI calls, e.g. source, root, tag, ierr etc. Typically 4 bytes, but ...

In order to model the running of your code, you may want to time a communication pattern.

How long does it take to start a message? What is the bandwidth for long messages? Would another MPI operator work better?

Fortran MPI data types include

<table>
<thead>
<tr>
<th>Fortran datatype</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL (single precision)</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
</tbody>
</table>

C data types include
Let's consider instead a ring program. It passes

FILE: ring.f

c ring program
c

c pass messages around a ring.
c Input: none.
c
\c See Chapter 3, pp. 41 & ff in PPMPI.
c
\c
\c program greetings
\c
\c include 'mpif.h'
\c
integer my_rank
integer p
integer source, source2
integer dest, dest2
integer tag, tag2
integer root
character*100 message,message2
character*10 digit_string
integer size
integer status(MPI_STATUS_SIZE)
integer ierr
integer i,nreps
real*8 startim,entim

c
    function
        integer string_len
        nreps = 10000
    c
        call MPI_Init(ierr)
    c
        call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
        call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
        startim = MPI_Wtime()
        do i=1,nreps
            call to_string(my_rank, digit_string, size)
            message = 'Greetings from process ' // digit_string(1:size)
+                 // '!' 
            if (my_rank.ne.p-1) then
                dest = my_rank+1
            else
                dest = 0
            endif
            if (my_rank.ne.0) then
                source = my_rank-1
            endif
else
    source = p-1
endif

tag = 0      ! message from even processors have an even tag
>tag2 = 1     ! messages from odd processors have an odd tag
root = 0

c
Note this solution only works if the total number of processors is even
Actually, it turned out to work !!
    if(my_rank.eq.2*(my_rank/2)) then ! if my_rank is even
        call MPI_Send(message, string_len(message), MPI_CHARACTER,
                    +         dest, tag, MPI_COMM_WORLD, ierr)
        call MPI_Recv(message2, 100, MPI_CHARACTER, source,
                    +         tag2, MPI_COMM_WORLD, status, ierr)
    else
        call MPI_Recv(message2, 100, MPI_CHARACTER, source,
                    +         tag, MPI_COMM_WORLD, status, ierr)
        call MPI_Send(message, string_len(message), MPI_CHARACTER,
                    +         dest, tag2, MPI_COMM_WORLD, ierr)
    endif
    call MPI_Get_count(status, MPI_CHARACTER, size, ierr)
    print*, 'my_rank=',my_rank
    write(6,101) message2(1:size), my_rank
101     format(' ',a,'   my_rank =',I3)
end do
entim = MPI_Wtime() - startim

c
call MPI_Finalize(ierr)
print*, ' elapsed time =',entim, '       my_rank=',my_rank
if (my_rank.eq.0)print*, 'number of reps =', nreps
end

Note: For each send there must be a matching receive.

An MPI_SEND is blocking. The program call MPI_SEND and waits till an acknowledgement from the matching MPI_RECV. This can be helpful in synchronizing code.

But notice we had to complicate things by writing if statements for odd and even rank processors. Else we would have had a hung code.

MPI is very rich. There are many ways to accomplish this same operation. e.g.

MPI_SENDRECV
FILE: ring2.f

c send messages right around a ring.
c
c This is simpler than ring.f
This is in that it uses the MPI_SENDRECV operator.
c
c Input: none.
c
c See Chapter 3, pp. 41 & ff in PPMPI.
c
    program greetings
    include 'mpif.h'

c integer my_rank
c integer p
c integer source, source2, right, left
c integer dest,dest2
c integer tag,tag2
c integer root
character*100 message,message2
character*10 digit_string
integer size
integer status(MPI_STATUS_SIZE)
c integer ierr
integer i,nreps
real*8 startim,entim

c
function
  integer string_len
  nreps = 10000
  
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
startim = MPI_Wtime()
do i=1,nreps
  call to_string(my_rank, digit_string, size)
  message = 'Greetings from process ' // digit_string(1:size) + '!
  if (my_rank.ne.p-1) then
    right = my_rank+1
  else
    right = 0
  endif
  if (my_rank.ne.0) then
    left = my_rank-1
  else
    left = p-1
  endif
  root = 0
c Send to the right -- receive from the left.
call MPI_SENDRECV(message, string_len(message), MPI_CHARACTER, + right, 0 , + message2, 100 , MPI_CHARACTER, + left, 0 ,
MPI.COMM_WORLD, status, err)
call MPI_Get_count(status, MPI_CHARACTER, size, ierr)
c    print*, 'my_rank=', my_rank
c    write(6,101) message2(1:size), my_rank

end do
entim = MPI_Wtime() - startim

call MPI_Finalize(ierr)
print*, 'elapsed time=', entim, ' my_rank=', my_rank
if(my_rank.eq.0) print*, 'nreps=', nreps
end

The syntax is as follows
1st argument, buffer (variable) to be sent --input
2nd argument, integer number of items in buffer (think vector) --input
3rd , data type--different Fortran and C bindings --input
4th , dest -- integer rank of processor to which message is sent input
5th tag1  -- integer message tag --input
6th receive buffer (name unchanged on output contains received data)
7th integer upper limit on number of received items --input
8th MPI data type --input
There is also an MPI_SENDRECV_REPLACE that uses the same send and receive buffer.

To get the basic MPI set completed, we'll also need global communications, non-blocking communications and ...
BASIC CONSIDERATIONS OF MESSAGE PASSING.

Compared to a memory access or to a computation, passing messages is expensive. Passing a message is more like accessing a hard drive. Just as a program that overflows RAM will "thrash", so a program that does fine-grained communication will run very slowly. It is very easy to write parallel programs that run more slowly than serial ones.

An add requires $O(1.e-9)$ secs in register
  $O(1.e-8)$ secs in L2 cache
  $O(1.e-7)$ secs in RAM

Other operations.
  $O(1.e-6$ to $1.e-7)$ secs for a subroutine call--or local MPI call such as MPI_PACK or MPI_UNPACK
  $O(1.e-4$ to $1.e-5)$ secs for an MPI_SEND message
  $O(1.-2$ to $e-3)$ secs access data from hard drive

So obviously, we want to make sure that when an add or multiply is be performed that we don't have to wait for a RAM, MPI, or hard drive fetch. So it makes sense to model communication. A simple model is
\[ T_c = (\text{time to start a message}) + (\text{bytes in a message}) \times (\text{time/byte}) \]

Or

\[ T_c = T_s + L \times t_b \]

where \( T_c \) is the time to start a message, typically \( 1.e-5 \) to \( 1.e-4 \) secs depending. \( t_b \) is the time to pass a byte is \( 1.e-8 \) secs for gigabit ethernet or myrinet. If we want to have the message time \( T_c \) in terms of the number of clock cycles or flops we would optimistically have

\[ T_c = 1.e4 + (\text{bytes in a message}) \times 200 \]

So we have to pass at least 1Kbyte-10Kbytes before the time start the message is less than half the message time.
MESSAGES. IT IS VERY EASY TO MAKE CODES RUN SLOWER IN PARALLEL THAN IN SERIAL. THE MOST COMMON BEGINNER ERROR IS TO PASS LOTS OF SMALL MESSAGES

So if we want to make a program slow, all we need do is pass lots of messages.

On the other hand, there is hope. If we can ration the number of messages to be received, then we have a good chance of getting a good parallel efficiency.

Thinking about data locality is an intrinsic part of getting good parallel code. It does seem a shame to burden users with it. But of course, it’s also what you have to do to get good performance in serial computing. After all, it’s less expensive to get data from another processor’s RAM than it is to get information from a local hard drive. So if your problem is too big to fit in RAM, you’ll probably get better performance by using more than one CPU allotment of RAM.

The discipline of considering data locality also enables more efficient serial code. For example, in serial computing, advertised flop rates are obtained
only when data in cache can be reused. Bus speed and cache size are often more important than CPU speed. For accessing data in RAM, we get a formula in clock cycles like

\[ T_a = 200 + (\text{number of bytes}) \times 50 \]

The computer tries to hide the 200 from you by bringing data into cache in blocks, but if you access the data in the wrong order (e.g., a matrix row wise instead of columnwise), you can get factors of 10 or more slow downs.

I’ll present an example for which accessing data from another allows more efficient use of cache memory, (perhaps) enabling superlinear speedup.

Reducing both the total "volume" and the "number" of messages will speed computations.

Successful parallel computations use "local" data for computations, requiring only periodic "global" refreshing, thereby keeping the global message volume small. Physical partitions are laid out to minimize the ratio
surface area/ volume.

Arctic bears weigh 1200 pounds, Florida bears weigh 250 pounds. Fully utilize RAM on one processor and just update the data on regions which are influenced by data resident on other processors.

Successful parallel computations find ways to overlap computation and communication. For instance, use non-blocking communications. We'll explore these later.

Not only does one minimize the volume of communication, but also the number of communications should be minimized. Short messages are "packed" into longer messages.

For example, we could pack integer values in a vector, floating point values in another vector, character values in a third vector. Three calls to MPI_PACK can pack the three vectors can be packed into a buffer of type MPI_PACKED
and sent in one message. Corresponding MPI_UNPACK calls after an MPI_RECV call can unpack the vectors. The initial integer vector can give instructions as to how many elements are to be unpacked.

Example. The following version of the Monte Carlo code packs up data into one message.

FILE: monte3.f

```
c
  Template for a Monte Carlo code.
c
  The root processor comes up with a list of seeds which it ships to all processors.
c
  In a real application, each processor would compute something and send it back. Here they compute a vector of random numbers and send it back.
c
  This version uses a single BCAST to distribute both integer and double precision data. It packs integer vectors as well as double precision data into an MPI_PACKED buffer.
c  This illustrates the use of MPI_PACK and MPI_UNPACK commands.
  
  program monte3

  include 'mpif.h'
```

integer my_rank
integer p
integer source
integer dest
integer tag
integer iseed, initseed, initvec(200)
integer status(MPI_STATUS_SIZE)
integer ierr
integer i, j, root
integer isize, position, kquad, nj
integer itemp(100), buf(4000)
real*8 ans(10), ans2(10), temp(100)
real*8 startim, entim

c

c
call MPI_Init(ierr)
c
call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
startim = MPI_Wtime()
c
do j = 1, 100
if (my_rank.eq.0) then
  iseed = 2
  initvec(1) = random(iseed)
  do i = 2, p
    initvec(i) = int(random()*1000)
  end do
  isize = (26 + nj + 2* kquad)
end if
ISIZE = (26 + nj + 2* KQUAD)
POSITION = 0
nj = 3
kquad = 4 ! actually these might have been read from a file
itemp(1) = isize
itemp(2) = kquad
do i=1,100
   temp(i) = 1. ! more realistically we would read from a file
end do
CALL MPI_PACK ( ITEMP, 2, MPI_INTEGER, BUF, 4000,
              +               POSITION, MPI_COMM_WORLD, IERR)
c This call packs the integer vector itemp of length 2 into buf
c starting at position 0. It increments position.
c --------------------------------------------------------------
c   See below to see how to unpack.
c   MPI_PACK and MPI_UNPACK are good for reducing the number of
c   total calls. These calls will allow us to pass multiple
c   messages for the latency of one. They are flexible in
c   that the length of the unpack can be part of the message.
c   The time of the calls to pack and unpack is not significant
c   compared to the time to pass a message.
c
   Disadvantage: On the Compaq AlphaServerSC the packing turned
c   out to be pretty loose, i.e., there were empty bytes. So combining
c   short messages would speed things, but long messages might get
c   enough longer that they would take more time.
c
--------------------------------------------------------------
CALL MPI_PACK ( initvec, 10, MPI_INTEGER, BUF, 4000,
              +               POSITION, MPI_COMM_WORLD, IERR)
CALL MPI_PACK ( TEMP, 100,
+                MPI_DOUBLE_PRECISION, BUF, 4000,
+                POSITION, MPI_COMM_WORLD, IERR)

endif

c pack up data into one message

root = 0
call MPI_BCAST(BUF, 4000, MPI_PACKED, 0,
+                MPI_COMM_WORLD, IERR)
call MPI_BARRIER(MPI_COMM_WORLD, ierr)
IF ( my_rank.NE.0 ) THEN
  POSITION = 0
C for the unpack, reset position to 0. The unpack order is
c the same as the pack order. But the order of arguments
c is changed.
c The call below unpacks integer vector itemp of length 2 from
c the BUF buffer.
c unpack itemp
    CALL MPI_UNPACK (BUF, 4000, POSITION, ITEMP, 2, MPI_INTEGER,
+                  MPI_COMM_WORLD, IERR)
    isize  = ITEMP(1)
kquad  = ITEMP(2)
c unpack initvec
    CALL MPI_UNPACK (BUF, 4000, POSITION, initvec, 10,
+                  MPI_INTEGER, MPI_COMM_WORLD, IERR)
    myseed = initvec(my_rank)
c unpack temp

    CALL MPI_UNPACK ( BUF, 4000, POSITION, TEMP, 100,
         +               MPI_DOUBLE_PRECISION, MPI_COMM_WORLD,IERR)
ENDIF

C-----------------------------------------------------------------------------------
C  Left out -- a body of code that does a bunch of particle tracking
C   stuff to produce the double precision vector ans
C-----------------------------------------------------------------------------------

call MPI_BARRIER(MPI_COMM_WORLD,ierr)
ans1 = rand(myseed)
do i=1,10
   ans(i) = rand() ! at least we initialize stuff to send back.
   ! But this call is something I had to change to get the code to
   ! run here.
end do

call MPI_REDUCE (ans,ans2, 10, MPI_DOUBLE_PRECISION,
             +               MPI_SUM, root, MPI_COMM_WORLD, ierr)
C-----------------------------------------------------------------------------------
C  Get the (sum of) data back
C  ans         -- arg1 -- message sent from each processor
C  ans2        -- arg2 -- result deposited on root -- out
C  10          -- arg3 -- length of message
C  MPI_DOUBLE_PRECISION --arg4 - data type
C  MPI_SUM      -- arg5 -- operation performed by reduce
C  root        -- arg6 -- reduce deposits answer on root
C  same on all processors
c MPI_COMM_WORLD -- arg7 -- all procs must have same communicator
ierr -- arg8 -- integer error--out (only in Fortran)

call MPI_BARRIER(MPI_COMM_WORLD,ierr)
if(my_rank.eq.0) then
    do some stuff to process and output ans2
endif
end do
entim = MPI_Wtime() - startim

call MPI_Finalize(ierr)
print*, ' elapsed time =', entim, '    my_rank=', my_rank
end
The MPI_PACK and MPI_UNPACK calls are quite fast compared to the
time to start a communication. The main drawback is that the
MPI_PACKED data form may include empty bytes, so that for example a
four byte integer might end up occupying 16 bytes in the MPI_PACKED
message. So the packing of messages will help with latency but not
bandwidth.

The most general MPI data type is the
MPI_TYPE_STRUCT -- the most general fixed MPI "derived data type"
allows multiple types of data entries.
This MPI data structure is loosely modeled on C structs (but allows other
data types than just integers). The MPI_TYPE_STRUCT may suffer from
the same loose packing as the MPI_PACKED structure. For the
MPI_STRUCT an MPI_Type_Commit is required as a declaration.

The MPI_Type_Commit operation requires more time than an MPI_PACK
but may be worthwhile if the same structure will be used repeatedly.
The following data types also require an MPI_Type_Commit as a
declaration.

MPI_VECTOR -- elements of a single type, allowing a stride.
MPI_INDEXED -- elements of a single type, with a variable indexed stride, e.g., to pass the upper triangular part of a matrix. MPI_TYPE_HVECTOR – if we want to compute the stride in bytes.

There is some expense in committing a data type, but it only occurs once, so is worthwhile if the same message type is to be frequently reused. Production codes I’ve seen used the VECTOR, INDEX, HVECTOR constructs. These are supported in MPI-2 IO. MPI_TYPE_STRUCT Is not.

Another "hacker" option.

Some MPI programmers just pack all their integer and double precision data into a single double precision vector and rely on type conversion. And plausibly you could also pack your character data into a double precision vector since you’re writing the “decode”.

Practical limitations of the
\[ T_c = T_s + L \times t_b \]

model are that it neglects:
1) noise in the system (e.g. processors have other tasks such as heartbeat, spare daemons) - \( T_s \) is sporadically large.
2) network saturation. Causes \( T_b \) to be sporadically large.
Both 1&2 can cause slower communications and motivate use of non-blocking sends and receives to allow overlapping of communication.

An advantage of the

\[ T_c = T_s + L \times t_b \]

model is we can model communication time via pen and paper before writing a large program. For instance we can conclude that laying out matrix data from square blocks will give total message lengths \( O(n /\sqrt{p}) \) for \( p \) processors. Where column blocks might give \( O(n) \)

EXERCISE:
Estimate \( T_c \) by repeatedly passing short messages, and \( L \) by passing
long messages. Note MPI comes with a good wall clock timer `MPI_Wtime()` which returns a double precision number and usually times in microseconds.

```c
startim = MPI_Wtime()
```

d stuff to time, e.g., 1000 repetitions of some MPI call.

```c
entim = MPI_Wtime() - startim
```

Then if we can track down all the communications and already know how long the computation will take in serial, then we can estimate parallel performance.

**EXERCISE:**
What parallel computation are you interested in? How long does it take in serial? Can you estimate how much communication is required? Can you predict parallel performance?

A typical 2-D parallel application may have communication volume $O(\sqrt{\text{volume on a processor}}) \times \log(\text{number of processors})$ where the
computations go as $O(\text{volume on a processor})$ So the total time is

$$T = O(\sqrt{V} \cdot \log(P)) + O(V)$$

and the parallel efficiency is

$$E = \frac{O(V)}{O(\sqrt{V} \cdot \log(P)) + O(V)}$$

which can be close to the ideal of one if $O(V)$ sufficiently large compared to $O(\sqrt{V})$. But slowness of communication puts a high constant on $O(\sqrt{V})$

Example:
Consider matrix vector multiplication. Suppose we'll multiply an $n \times n$ matrix $A$ times an $n$-vector $x$ and return the vector $x$ to the root processor. Assume that $A$ is already distributed to all processors (communicating $A$ to all processors would require more time than computing $Ax$ on one processor). In particular, assume that $A$ is distributed with $n/p$ columns per processor.
A = \[ A_1 | A_2 | \ldots | A_p \]
so that \( A_i \) has \( \frac{n}{p} \) columns. Partition \( x \) with \( \frac{n}{p} \) elements per block
\( x = [x_1 | x_2 | \ldots | x_p] \)

Then

\[
A \times x = A_1 \times x_1 + A_2 \times x_2 + \ldots + A_p \times x_p
\]

can be performed as
1) scatter \( x_i \) to processor \( i, i=1:p-1 \) (MPI_SCATTER)
2) In parallel perform \( w_i \leftarrow A_i \times x_i, i=1,p-1 \) -- computations on each node.
3) perform an MPI_REDUCE to add all the \( w_i \) to get \( x \) on the head node.

Communication costs are mainly for the reduce, which is passing a vector of length \( n \), \( \log(p) \) times, i.e., \( 8n \log(p) t_b \) compared to \( 2n^2 \frac{n}{p} \) flops for each of the \( A_i \times x_i \)

so if \( t_b \) is flops/byte (how many flops can be performed in the time it takes to pass an additional byte of a message) the parallel efficiency should be
\[ E = \frac{2n^2/p}{2n^2/p + 8n\log(p)\cdot t_b} \]

\[ = \frac{n/p}{n/p + 4\log(p) \cdot t_b} \]

E will be near one if \( n/p >> 4 \cdot \log(p) \cdot t_b \)

i.e., we should have a problem size (for good parallel efficiency with column blocking)

\[ n >> p \cdot (4 \cdot \log(p) \cdot t_b) \]

where the DOD tries to keep \( t_b \) at about 200.

For the column blocking scheme analysed so far, the total volume of communication is \( O(n) \). We can reduce that to parallel \( O(n/\sqrt{p}) \) by partitioning the matrix into square blocks of size \( n/\sqrt{p} \), so that the work per processor is the same. Then we get

\[ E = \frac{2n^2/p}{2n^2/p + 4n\log(p)\cdot t_b/\sqrt{p}} \]
i.e., to get $E$ near one we should have (for the case that communications don't interfere with each other, and that the matrix in the matrix vector multiply is dense)

$$n >> \sqrt{p} \times (2 \times \log(p) \times t_b)$$

To keep $E$ fixed as the number of processors grows we see we need to grow the problem size $n$. In this case, as we hold the problem size $n/\sqrt{p}$ per processor fixed, the communication only grows as $\log(p)$ so we would say the computation is scalable.
COLLECTIVE COMMUNICATION

As we’ve already seen in some examples -- Having defined a communicator, (set of processors) some common patterns of communication have special MPI commands.

One example is MPI_BCAST, which broadcasts a message from a specified processor to all other processors. We could implement this (as in the prototype Monte Carlo code) by

```fortran
if (my_rank.ne.0) then
    do i = 1,p - 1
        call MPI_SEND
    end do
else
    call MPI_RECV
endif
```

but this would require a time proportional to the number p of processors. Whereas, if we implement a "fan-out" algorithm, the cost would go like log p.
By using the MPI_BCAST program, we save the bother. The standard mpich library implements "fan-out". And typically, on a given architecture, this is an algorithm someone has optimized.

call MPI_BCAST( ) same arguments on each processor.

Another common operation is a gather. Each processor contributes some entries to a vector.

call MPI_GATHER( )

call MPI_SCATTER( )

Instead of broadcasting an entire vector, send the first $k$ entries to a first processor, the next $k$ a second, etc. For example, in the monte program, we could have a vector of seeds, so instead of having a loop with matched sends and receives, the root processor would have a single MPI_SCATTER call. Each of the other nodes also would require the same call. If each processor is to get a different number of entries, we can use
call MPI_SCATTERV(                    )

To collect a different number of entries from each processor

call MPI_GATHERV

Another useful operation is a reduce.

call MPI_REDUCE(                    reduce_op)

If the reduce_op is addition, a sum of entries (or a sum of vectors) would be deposited on the root processor. Other reduction operations are min, max, max_loc, min_loc, and multiplication. Here's the Monte code redone using an MPI_SCATTER and an MPI_REDUCE

FILE: monte2.f

c Template for a Monte Carlo code.
c

c The root processor comes up with a list of seeds
c which it ships to all processors.
In a real application, each processor would compute something and send it back. Here they compute a vector of random numbers and send it back.

This version uses a scatter to distribute seeds and a reduce to get a sum of distributed data

```
program monte2

include 'mpif.h'

integer my_rank
integer p
integer source
integer dest
integer tag, root
integer iseed,initseed,initvec(200)
integer status(MPI_STATUS_SIZE)
integer ierr
integer i
real*8 ans(10), ans2(10)
real*8 startim,entim

function
    integer string_len

    call MPI_Init(ierr)
```
call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)

if (my_rank.eq.0) then
    print*, 'input random seed'
    read*, iseed
    initvec(1) = random(iseed)
    do i = 2, p
        initvec(i) = int(random() * 1000)
    end do
endif

--call MPI_BARRIER(MPI_COMM_WORLD,ierr)!
!What difference is there?
startim = MPI_Wtime()

root = 0
    call MPI_SCATTER(initvec, 1, MPI_INTEGER,
                      myinit, 1, MPI_INTEGER,
                      root, MPI_COMM_WORLD, ierr)

Scatter message consisting of
initvec -- arg 1 message sent
1 -- arg 2, length of message to each processor
MPI_INTEGER -- arg 3, type of data sent
myinit -- arg 4, message received on each processor -- output

SimpleMPI - Page 46
c 1       -- arg 5, length of message received on each processor
c MPI_INTEGER -- arg 6, type of data received
c root       -- arg 7, must be identical for all processors
c origin of message
c MPI_COMM_WORLD -- arg 8, must be same on all processors.
c ierr       -- error info, only in Fortran -- output
c This call may or may not be blocking, not specified in standard.
c----------------------------------------------------------------------
c-----------------------------------------------------------------------
c Left out -- a body of code that does a bunch of particle tracking
c stuff to produce the double precision vector ans

call MPI_BARRIER(MPI_COMM_WORLD,ierr)
ans1 = rand(myseed)
do i=1,10
   ans(i) = rand() ! at least we initialize stuff to send back.
end do

call MPI_REDUCE (ans,ans2, 10, MPI_DOUBLE PRECISION,
+                MPI_SUM, root, MPI_COMM_WORLD, ierr)
c----------------------------------------------------------------------
c Get the (sum of) data back
c ans       -- arg1 -- message sent from each processor
c ans2      -- arg2 -- result deposited on root -- out
c 10        -- arg3 -- length of message
c MPI_DOUBLE PRECISION --arg4 - data type
c MPI_SUM    -- arg5 -- operation performed by reduce
c root       -- arg6 -- reduce deposits answer on root

SimpleMPI - Page 47
same on all processors

MPI_COMM_WORLD -- arg7 -- all procs must have same communicator

eerr -- arg8 -- integer error--out (only in Fortran)

c------------------------------
call MPI_BARRIER(MPI_COMM_WORLD,ierr)
if(my_rank.eq.0) then
do some stuff to process and output ans2
endif
entim = MPI_Wtime() - startim
call MPI_Finalize(ierr)
print*, ' elapsed time =',entim, ' my_rank=',my_rank
end

c
cc

SimpleMPI - Page 48
An MPI_ALLREDUCE will deposit the "sum" on all processors in the communicator.

MPI_ALL_GATHER will perform the “gather” onto all processors in the communicator.

Example
Matrix vector multiplication, use row blocks, bcast vector to all. then gather the answer use column blocks, scatter x, use reduce to get y on root.

Example, gemvt in the next section.

One complication of global communications: It is not specified in the standard whether they should be blocking. If you want to ensure that a global communication is completed before the program proceeds, a safe way is

    call MPI_BARRIER(communicator)

Collective communications valid only within a communicator.

Multiple Communicators
MPI defines communicators. A motivation is to provide a context so that library functions can be secure and non-conflicting. Global communications are easy to use and efficient. They work only within a communicator. One way to provide multiple communication groups is to provide multiple communicators.

**MPI_COMM_SPLIT**

allows a communicator to be split into sub-communicators. We can define structure to a communicator by using topologies. Example, Cartesian topologies.

Example.
FILE: p2gemv.f
Input: matrix A, vector x, currently hardwired.
Output: results of calls to various functions testing topology creation

This one will do paired matrix vector multiplications using the square topology (gemvt).

Limitations of code so far:
1) each processor has to have the same size matrix
2) block size kb is hardwired at 100
3) the number of processors total must be a perfect square.

Note: Assumes the number of processes, p, is a perfect square

See Chap 7, pp. 121 & ff in PPMPI

PROGRAM PGEMVT
INCLUDE 'mpif.h'
integer lda
parameter (lda=5000)
integer       p,m,n,k,nb,i,j
real          preal
real          a(lda,lda),x(lda),y(lda),z(lda),w(lda)
integer       my_rank
integer       q
integer       grid_comm
integer       dim_sizes(0:1)
integer wrap_around(0:1)
integer reorder
integer coordinates(0:1)
integer my_grid_rank
integer grid_rank
integer free_coords(0:1)
integer row_comm
integer col_comm
integer row_test
integer col_test
integer ierr
real*8 entim, start

C
C
m = 4000  ! these are the local matrix size
n = 4000
k = 40
reorder = 1
call MPI_INIT( ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, p, ierr )
call MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr )
C
preal = p
q = sqrt(preal)
C
dim_sizes(0) = q
dim_sizes(1) = q
wrap_around(0) = 0
wrap_around(1) = 0
call MPI_CART_CREATE(MPI_COMM_WORLD, 2, dim_sizes,
+     wrap_around, reorder, grid_comm, ierr)

C

    call MPI_COMM_RANK(grid_comm, my_grid_rank, ierr)
    call MPI_CART_COORDS(grid_comm, my_grid_rank, 2,
                         +     coordinates, ierr)

C

    call MPI_CART_RANK(grid_comm, coordinates, grid_rank,
                        +     ierr)

C

    free_coords(0) = 0
    free_coords(1) = 1

    call MPI_CART_SUB(grid_comm, free_coords, row_comm, ierr)
    call MPI_COMM_SPLIT(MPI_COMM_WORLD, coordinates(0), my_rank,
                        +     row_comm, ierr)
    if (coordinates(1) .EQ. 0) then
        row_test = coordinates(0)
    else
        row_test = -1
    endif
    call MPI_BCAST(row_test, 1, MPI_INTEGER, 0, row_comm, ierr)

C

    print*, 'after first cartsub', row_test, my_rank
    free_coords(0) = 1
    free_coords(1) = 0
    call MPI_BARRIER(MPI_COMM_WORLD, ierr)

C

    call MPI_CART_SUB(grid_comm, free_coords, col_comm, ierr)
    call MPI_COMM_SPLIT(MPI_COMM_WORLD, coordinates(1), my_rank,
                        +     col_comm, ierr)
    if (coordinates(0) .EQ. 0) then
col_test = coordinates(1)
else
    col_test = -1
endif

call MPI_BCAST( col_test, 1, MPI_INTEGER, 0, col_comm,ierr)

c     print*,'after second cartsub',col_test,my_rank

c     call MPI_BARRIER(MPI_COMM_WORLD,ierr)

c
A more general matrix input would be useful

The global matrix has entries a(i,j) = i-j
the global x vector has entry x(i) = i

c     print*,'my_rank,9 and 1',my_rank,coordinates(0),coordinates(1)
do j=1,n
    do i=1,m
        a(i,j) = coordinates(0)*m+i - (coordinates(1)*n+j)
c           print*,'a(i,j),my_rank=',a(i,j),i,j,my_rank
    end do
y(i) = 0.0
end do
do i=1,m
    x(i) = coordinates(0)*m + i
    print*,'x('','i','')='x(i),'my_rank='my_rank
w(i) = 0.0
end do
if(my_rank.eq.0) then
    start = MPI_Wtime()
endif
do ii=1,10
Each processor belongs to a column communicator, so will use that
communicator to help with the global matrix vector product $x^T A_i$
looping through the column blocks $A_i$ for the local matrix $A$  

```plaintext
k_b = n/k
```

```plaintext
c    print*: 'kb =', kb
do i=1,m
    x(i) = coordinates(0)*m + i + ii/10.
c    print*: 'x('i,')=', x(i), 'my_rank=', my_rank
    w(i) = 0.0
end do
do i=1,n/kb
    call sgemv('T', m, k_b, 1.0, a(1, (i-1)*k_b+1),
                  lda, x, 1.0, y((i-1)*k_b+1), 1)
call MPI_ALLREDUCE(y((i-1)*k_b+1), z((i-1)*k_b+1),
                  k_b, MPI_REAL, MPI_SUM, col_comm, ierr)
call MPI_BARRIER(MPI_COMM_WORLD, ierr)
call MPI_BCAST(z((i-1)*k_b+1),
                k_b, MPI_REAL, 0, col_comm)
call sgemv('N', m, k_b, 1.0, a(1, (i-1)*k_b+1),
                  lda, z((i-1)*k_b+1), 1, 1.0, w, 1)
end do
c  Take care of extra columns (for the case that k does not divide n exactly
    call sgemv('T', m, n-(n/kb)*k_b, 1.0, a(1, (i-1)*k_b+1),
                  lda, x, 1.1.0, y((i-1)*k_b+1), 1)
call MPI_ALLREDUCE(y((i-1)*k_b+1), z((i-1)*k_b+1),
                  n-(n/kb), MPI_REAL, MPI_SUM, col_comm, ierr)
call sgemv('N', m, n-(n/kb)*k_b, 1.0, a(1, (i-1)*k_b+1),
                  lda, z((i-1)*k_b+1), 1, 1.0, w, 1)
c  Column communicator operations are done, next do the row communicator
```
operation to sum up the local Aw's to get the global x

call MPI_BARRIER(MPI_COMM_WORLD,ierr)

do j=1,n
    print*, 'z(',j,')=',z(j),my_rank
    print*, 'y(',j,')=',y(j),my_rank
end do

call MPI_ALLREDUCE(w,x,m,
+ MPI_REAL,MPI_SUM,row_comm)
call MPI_BARRIER(MPI_COMM_WORLD,ierr)

After the scatter reduce, each row communicator has the necessary local x to start over (or to do local updates)
    end do
call MPI_BARRIER(MPI_COMM_WORLD,ierr)
if(my_rank.eq.0)then
    entim = MPI_Wtime()-start
    print*,entim
    do i=1,m
        print*, 'x(',i,')=',x(i)
    end do
endif
if(my_rank.eq.3)then
    entim = MPI_Wtime()-start
    print*,entim
    do i=1,m
        print*, 'x(',i,')=',x(i)
    end do
endif

call MPI_FINALIZE(ierr)
Advanced Point to Point

Yesterday we considered collective communication. It has the advantage that the same call can be used on all processors in a communicator. And
the communications take advantage of “roll-in”, “roll-out” patterns of communication so that the time to communicate to \( p \) processors goes like \( \log(p) \). If we need different groups of processors, we can split an existing communicator into sub communicators. Then on each smaller communicator, we can use collective communication routines.

REMEMBER: collective communications may not be blocking. If we want to make sure that the communication has completed, put an MPI_BARRIER call after the collective communication call.

The first day we did point to point communication. If we want to communicate to \( p \) processors, these are inefficient, (time is \( O(p) \)) but they are the method of choice for communicating to a few processors.

So far, we considered some ways to do communication between adjacent processors

paired MPI_SENDs and MPI_RECVs (sending message around a ring)
if (my_rank is even) then
    call MPI_Send(message, string_len(message), MPI_CHARACTER, 
                  dest, tag, MPI_COMM_WORLD, ierr)
    call MPI_Recv(message2, 100, MPI_CHARACTER, source, 
                   tag2, MPI_COMM_WORLD, status, ierr)
else
    call MPI_Recv(message2, 100, MPI_CHARACTER, source, 
                   tag, MPI_COMM_WORLD, status, ierr)
    call MPI_Send(message, string_len(message), MPI_CHARACTER, 
                   dest, tag2, MPI_COMM_WORLD, ierr)
endif

complicated. The wrong order would cause the code to hang.

or simpler MPI_SENDRECV

c Send to the right -- receive from the left.
    call MPI_SENDRECV(message, string_len(message), MPI_CHARACTER, 
                     right, 0 ,
                     message2, 100 , MPI_CHARACTER, 
                     left, 0 ,
                     MPI_COMM_WORLD, status, err)
    call MPI_Get_count(status, MPI_CHARACTER, size, ierr)

which at least won’t block as easily. It is blocking. Meaning that code lines after this will not be executed until after this operation is completed.
We may prefer nonblocking communications. They allow us to have the computer do something else while the communication completes. Also we don’t have to worry (so much) about the order of sends and receives.

One way is to do ISENDs and IRECVs.

FILE: nonblock.f

c  ring program
  
c  pass messages "right" around a ring.
  
c
c Input: none.
c
This version uses nonblocking sends. This is convenient in that we don't have to keep track of the order of the sends.
Buffer overflow would be possible, so this method is not "safe"

See Chapter 3, pp. 41 & ff in PPMPI.

program greetings
c
    include 'mpif.h'
c
    integer my_rank
    integer p
    integer source, source2
    integer dest, dest2
    integer left, right
    integer tag, tag2
    integer root
    character*100 message, message2
    character*10 digit_string
    integer size
    integer status(MPI_STATUS_SIZE)
    integer ierr, request
    integer i
    real*8 startim, entim
c

SimpleMPI - Page 61
function  
    integer string_len

    call MPI_Init(ierr)

    call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
    startim = MPI_Wtime()
    do i=1,1
        call to_string(my_rank, digit_string, size)
        message = 'Greetings from process ' // digit_string(1:size)
+                           // '!
        if (my_rank.ne.p-1) then
            right = my_rank+1
        else
            right = 0
        endif
        if (my_rank.ne.0) then
            left = my_rank-1
        else
            left  = p-1
        endif
        tag = 0      ! message from even processors have an even tag
        tag2 = 1     ! messages from odd processors have an odd tag
        root = 0

        call MPI_ISEND(message, string_len(message), MPI_CHARACTER, 
+        right, 0, MPI_COMM_WORLD, request ,ierr)
        call MPI_IRECV(message2, 100, MPI_CHARACTER, left, 

SimpleMPI - Page 62
call MPI_Wait(request, status, ierr)
call MPI_Get_count(status, MPI_CHARACTER, size, ierr)
c        print*, 'my_rank=', my_rank
        write(6,101) message2(1:size), my_rank
101    format(' ',a,' my_rank=',I3)
end do
entim = MPI_Wtime() - startim
c    call MPI_Finalize(ierr)
    print*, 'elapsed time=', entim, ' my_rank=', my_rank
end
Programming was simplified, but alas the times were not as good. Where the MPI_SENDRECV or the times for paired MPI_SEND and MPI_RECV calls were around 5 micro seconds (quadrics switch), the times for the MPI_ISEND and MPI_IRECV pair was 17 micro seconds.

Of course, if we could do some computations in the meantime, this might be okay. (On the quadrics switch) It turned out there is a way to do nonblocking sends and receives faster, by persistent requests.
FILE: persist3.f

ring program

pass messages "right and left" around a ring.

Input: none.

This version uses nonblocking sends. This is convenient in that we don't have to keep track of the order of the sends. Buffer overflow would be possible, so this method is not "safe"

See Chapter 3, pp. 41 & ff in PPMPI.

program greetings

    include 'mpif.h'

    integer my_rank
    integer p
    integer source, source2
    integer dest, dest2
    integer left, right
    integer tag, tag2
    integer root
    character*100 message,message2,message3
    character*10 digit_string
integer size, size2
integer status(MPI_STATUS_SIZE), status2(MPI_STATUS_SIZE)
integer ierr, request, request2, request3, request4
integer i, nreps
real*8 startim, entim

c

function
integer string_len

c

nreps = 10000
call MPI_Init(ierr)

c

call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
startim = MPI_Wtime()
call to_string(my_rank, digit_string, size)
message = 'Greetings from process ' // digit_string(1:size)
+ '!
if (my_rank.ne.p-1) then
  right = my_rank+1
else
  right = 0
endif
if (my_rank.ne.0) then
  left = my_rank-1
else
  left = p-1
endif

c This call initiates the required sends and receives. Note all persistent requests are
c nonblocking.
call MPI_SEND_INIT(message, string_len(message), MPI_CHARACTER, +
right, 0, MPI_COMM_WORLD, request ,ierr)
call MPI_RECV_INIT(message2, 100, MPI_CHARACTER, left,
+          0, MPI_COMM_WORLD, request2 ,ierr)
call MPI_SEND_INIT(message, string_len(message), MPI_CHARACTER, +
left, 1, MPI_COMM_WORLD, request3 ,ierr)
call MPI_RECV_INIT(message3, 100, MPI_CHARACTER, right,
+          1, MPI_COMM_WORLD, request4 ,ierr)
c One easy refinement -- put the requests as req(1), req(2), req(3), req(4)
c Then the start calls become CALL MPI_STARTALL(4, req,ierr)
c and the wait call become CALL MPI_WAITALL(4, req,ierr)
do i=1,nreps
   call MPI_START(request,ierr)
call MPI_START(request2,ierr)
call MPI_WAIT(request,status,ierror)
call MPI_WAIT(request2,status,ierror)
c call MPI_Get_count(status, MPI_CHARACTER, size, ierr)
call MPI_START(request3,ierr)
call MPI_START(request4,ierr)
call MPI_WAIT(request3,status2,ierror)
call MPI_WAIT(request4,status2,ierror)
c call MPI_Get_count(status2, MPI_CHARACTER, size2, ierr)
c print*, 'my_rank=', my_rank
write(6,101) message2(1:size), my_rank
write(6,101) message3(1:size2), my_rank
101 format( ',a,' my_rank =',i3)
end do
entim = MPI_Wtime() - startim
call MPI_Finalize(ierr)
print*, 'elapsed time =', entim, '      my_rank=', my_rank
if(my_rank.eq.0) print*, 'number of reps =', nreps
end

The refined version returns to the 5 microsecond time on the HP SCs. And is (as are all persistent requests) nonblocking.

On the same machine I was able to get lower latencies with the shmem library. This library lets us use one-sided communication. Shmem is available for SGIs, Crays, and machines with a quadrics switch. Latency there was about 3 microseconds with a barrier between repeated pings.

Shmem was one of the motivations for MPI-2.
MPI is a rich library. Son of MPI (MPI-2) is even richer. The standard was published around 1995. Only now are complete implementations starting to appear. Some of its functionality is available on the cluster here.

A primary motivation of MPI-2 was to include features available in other libraries. (A reservation to MPI-2. It may make MPI so feature rich that like Ada or PL-2 no one has the energy to use it all, or to provide implementations that do all MPI well.)

Some rationales for MPI-2

PVM could spawn processes and provide links between already existing parallel computations. PVM provides standard interfaces for heterogeneous networks (networks consisting of more than one machine architecture – for example Sun Solaris, Linux , and IBM AIX machines)

Shmem efficiently imitates shared memory by allowing computers to get and put onto other processors.

I/O. MPI ignored I/O – global and local access to files on hard drives. This is a significant part of making parallel codes efficient.
So before we’ve completely forgotten about passing messages between adjacent processors, consider one sided communications.

An example of one-sided communication for a “ping”.

Code fragment (from MPI-The Complete Reference Vol. 2)

```fortran
Call MPI_TYPE_SIZE(MPI_REAL, sizeofreal, ierr)
Call MPI_WIN_CREATE(Blocal, m*sizeofreal, sizeofreal, &
                     MPI_INFO_NULL, comm, win, ierr)

! returns the communication window win.
Call MPI_WIN_FENCE(0,win,ierr)
Do i=1,m
   J = mapLocal(i)/p
   K = MOD(maplocal(i),p)  ! mod function
   CALL MPI_GET(Alocal(i), 1, MPI_REAL, j, k ,1 MPI_REAL, win, ierr)
END DO
Call MPI_WIN_FENCE(0, win, ierr)
CALL MPI_WIN_FREE(win,ierr)
```

The MPI_GETs are not blocking. The MPI_WIN_FENCE enforces synchronization.

How well does this work?
I compared times for doing a “ping” with (HP AlphaServer SC with a quadrics switch).

78.8 microSecs (averaged over 10K repetitions)  
for  
MPI_Fence  
MPI_Put  
MPI_Fence  
------------------------------------------------------

44.34 microSecs (averaged over 10K repetitions)  
for  
MPI_Fence  
MPI_Put  
------------------------------------------------------------

5.47 microSecs (averaged over 10K repetitions)  
MPI_Put  
----------------------------------------------------------
So in this case the expensive operation is the synchronization.

Results of comparing various “pings” on a quadrics switch on The SC Alpha Server SC40 at ERDC – one of the five fastest machines in the world is of this type. If you develop good code you can benchmark on the 3000 CPU machine at Pittsburgh SuperComputing.

Comparisons of latency $T_s$ for various calls on an SC40

- Shmem: 3 micro secs (Dick Foster’s library With synchronization Exists here on the p690)

The rest are for calls developed by the MPI committee

- Send Recv paired: 5 micro secs (blocking)
- Send Recv as one command: 5 micro secs (blocking)
- MPI_GET one sided with no synchronization: 5 micro secs
- Persistent blocking: 5 micro secs
Nonblocking with Wait after 17 micro secs
One sided
    with synchronization 44 micro secs

I haven’t had a chance to time all these here.

Local results.

On Henry2,
For MPI_SENDRECV latency is about 40 micro secs (80 microseconds for each processor to send and receive one message)

The observed bandwidth for MPI_SENDRECV was about 40 Mbytes per second (40 million bytes sent and 40 million bytes received in 2 seconds). This was from sending and receiving 10 messages each of size 4 Mbytes).

So, in this morning’s experiments, time for a message on henry2 is estimated as

\[ T_c = 4.0 \times 10^{-5} + \text{bytes} \times (2.5 \times 10^{-8}) \text{ seconds} \]
Assuming one flop requires 1.e-9 seconds, the message cost is equivalent to

\[ T_c = 4.5\times10^4 + (\text{number of bytes}) \times 25 \text{ flops} \]

Or

\[ T_c = 4.5\times10^4 + (\text{number of integers}) \times 100 \text{ flops} \]

Or

\[ T_c = 4.5\times10^4 + (\text{number of doubles}) \times 200 \text{ flops} \]

The message length for which half of the time is spent waiting is about 
\[ 4.5\times10^4/25 = 1800 \text{ bytes} = 450 \text{ integers} = 225 \text{ doubles}. \]

So unless your messages are at least this long, most of the message passing time will be in the waits for the messages to start.

For messages longer than this we would probably not use MPI_PACKED format (as it may not be tightly packed so wastes bandwidth).
For messages shorter than this, combining messages into MPI_PACKED is probably a good idea.

I/O in parallel computation.

We’ve concentrated on communication between processors. In practice, one of the main communication problems is in disk I/O. The program and initial data are written somewhere on hard drives and must be transported to the computational nodes. Results must be written to disk.

Again, we can give an initial model for the time for “passing a message” as

\[ T_c = T_s + (\text{length of message in bytes}) \times T_b \]
This comes out to (in seconds)

\[ T_c = 1.0 \times 10^{-2} + (\text{length of message in bytes}) \times 5.0 \times 10^{-8} \]

Or in flops,

\[ T_c = 1.0 \times 10^{7} + (\text{length of message in bytes}) \times 50 \]

(assuming disk access time of 10 milliseconds and read write bandwidth of 20 Mbytes/sec, and that a flop take 1.0 \times 10^{-9} seconds).

1) A main difference here is the disk access time.

2) The other main complication: most parallel computers have a single image file system. This is convenient in that we can read and write from any processor to a file which will be globally accessible.

But since the file is globally accessible, reads and writes are intrinsically serial. If many different users are reading and writing to the same file system, there may be a good deal of contention. (In contrast,
communication between processors is relatively free of contention).

Due to 1) and 2), “latency hiding” is even more important than in usual message passing.

The main trick to “latency hiding” is caching I/O data in RAM. This is accomplished either by the system or by the user.

**System caching of data**

The system uses RAM on either the file server node or on the computational node, or both as a buffer (cache). This is invisible to the user until the capability is taken away.

Generally, it’s a good idea to read and write relatively large blocks of data. One code I saw recently attempted to minimize the total number of bytes read. Each processor read a few bytes and used the information from those bytes to jump to another point in the file and then read a few more bytes.
The code I/O times were proportional to the number of processors used. For 128 processors, I/O times went to ten hours. This was with a high performance file system that could read or write 80 Mbytes/sec by striping its writes across a number of RAID boxes. That (PFS) file system accomplishes a high bandwidth by establishing a direct connection from the computational node to the file server node. The “chatter” to set up the direct connection is a bit expensive. 700 connections can be made per second. If each connection delivers only one number, then 700 numbers per second are read or written (as opposed to the peak rate of 10Mbytes?)

Then to get a permanent record we have to write out results.

Writes should be made periodically. If many processors are used for long enough periods of time, it’s likely one will fail, so codes to be truly scalable should occasionally back up data in such a way that the code can be restarted. This is called a “checkpoint restart”.

A typical write rate to a RAID box (collection of four or five disk drives with a parity check disk so that any one disk can go out without a loss of data) may have a rate of 20 Mbytes/sec.
The most common file system is NFS (network file system). It is mature and robust. It accomplishes RAM buffering. So the user can for example, write a number at a time to a file and not see a drastic performance hit.

It has some problems with scalability. It’s the method used on the cluster file system on henry2.

NFS allows memory mapping and file locking, among other features. Since NFS is so universal, MPI-2 I/O operations are likely to work.

Because modern versions of NFS cache data, write (and often read) rates are fairly independent of write size. Using NFS instead of PFS, the I/O time was reduced from ten hours to ten minutes.

In practice, the write is to cache (local RAM) or to caches on the file server node. As soon as data has been copied to a RAM buffer, the write or printf statement returns control to the program and executes the next statement.

A downside of caching data is that cached data is not yet actually saved to hard drive so can be lost. In order to ensure data is saved we need to demand a disk synchronization. Calling the C function fdatasync is one
way to ensure data have been written.

User Handling of I/O

The most common user optimization is to lump data so that it can be transferred in a few blocks, with rather little synchronization between computational nodes.

For portability, users should read and write in large chunks. In Fortran 90, one can specify records and sizes and can fill the records before writing them. In C, fopen statements can be immediately followed by setvbuf statements specifying large buffer sizes. Compiler options sometimes allow enlarged buffers.

MPI-2 IO allows nonblocking reads and writes. So the user can start a read or a write, and then at some other point in the code (just before the data is used) specify a test that the read or write is complete.

For example,
CALL MPI_FILE_OPEN( ) returns integer file handle fh
CALL MPI_FILE_SET_VIEW( ) tells how to look at file
CALL MPI_FILE_IREAD( fh, buf1, bufsize, MPI_REAL, req, ierr ) would prefetch data and then the code could do some things and eventually an …

…

CALL MPI_WAIT( req, status, ierr ) ! synchronization call
CALL MPI_FILE_CLOSE( fh, ierr ) ! release the file handle.

Question: how to use MPI-2 IO calls when they are file system dependent? One possibility is to write your intent in MPI-2 calls and then make system dependent alternatives.

User problems caused by system caching – solution to have one processor make the reads and writes and then communicate to others by MPI?

Problem: If many processors access the same file there can be delays since files may not be actually be written to disk (the data is lurking somewhere in a cache. Another process can’t actually access the disk till the write is complete).

One user accomplished check point restarts by writing to a file from one
processor, then handing off the read to another processor, looping through all processors. The hand-offs turned out to be slow, requiring about 30 seconds each to flush the caches and lock the files. Again this was a “feature” of the high performance file striping file system. As the number of processors grew, the hourly checkpointing occupied an excessive amount of time (e.g., half an hour when he used 64 processors).

In his case, he found it faster to designate one processor as the communication node. He used MPI messages to transfer data to the root processor, which then accomplished the write to disk.

Some other User tricks.

Recall that using the global file system is causing contention. Essentially everyone is trying to read and write to the same file system, so delays can result. Users can write to the local file system and then have a background copy to the global file system. Since access to the local file system is lost when the parallel job completes, the parallel job can’t complete till copies from the local to global file system have been made. A portability issue is that sizes of local file systems are highly variable.
It is possible to have pre and post I/O operations. For example, one processor can assemble the input for each processor into an individual file. Then all processors can just read their own file. Or each processor can just write to its own file, and then later on a postprocessing job combines those files into one data file.

**Portability Issues Not Addressed**

Various machine resources can be exceeded. Buffer sizes. Number of allowed open messages. Disk sizes on local hard drives.

Also we haven’t done much with debugging and profiling.

**Use of Parallel Libraries**

MPI is just one of the available parallel libraries. It was designed to allow
development of other parallel libraries (partly by having messages that do not interfere with messages in other communicators). Some other libraries are:

Lapack and BLAS (Basic Linear Algebra Subroutines) give efficient matrix computations on single processors. These are part of the Intel Math library (which exists on henry2).

SCALAPACK is a parallel version of LAPACK. At least part of SCALAPACK is in the Intel Math Library.

Some other libraries which should be installed are SuperLU (uses “direct” LU decompositions to solve Ax=b, for the case that A is sparse).

PARPACK, used to find eigenvalues of sparse matrices.

pARMS – iterative solution sparse matrix equations.

PETSc – extensible package for scientific computation.

For some others, see the NERSC parallel repository.

Most of these are easy to install. If you need them quickly (before
January) it may be worthwhile to install them yourself.

You may find that some licensed software will be useful to you and others. If so, please tell us.

Some packages being considered are

Abaqus, Ansys, NASTRAN?

All of these are finite element codes which can be applied to a number of physical situations. And TotalView as a debugger.

Conclusions?
Please send your ideas and concerns to gwhowell@ncsu.edu or eric_sills@ncsu.edu