MPI Part 1

* References:

  Using MPI. Gropp, Lusk Skjellum

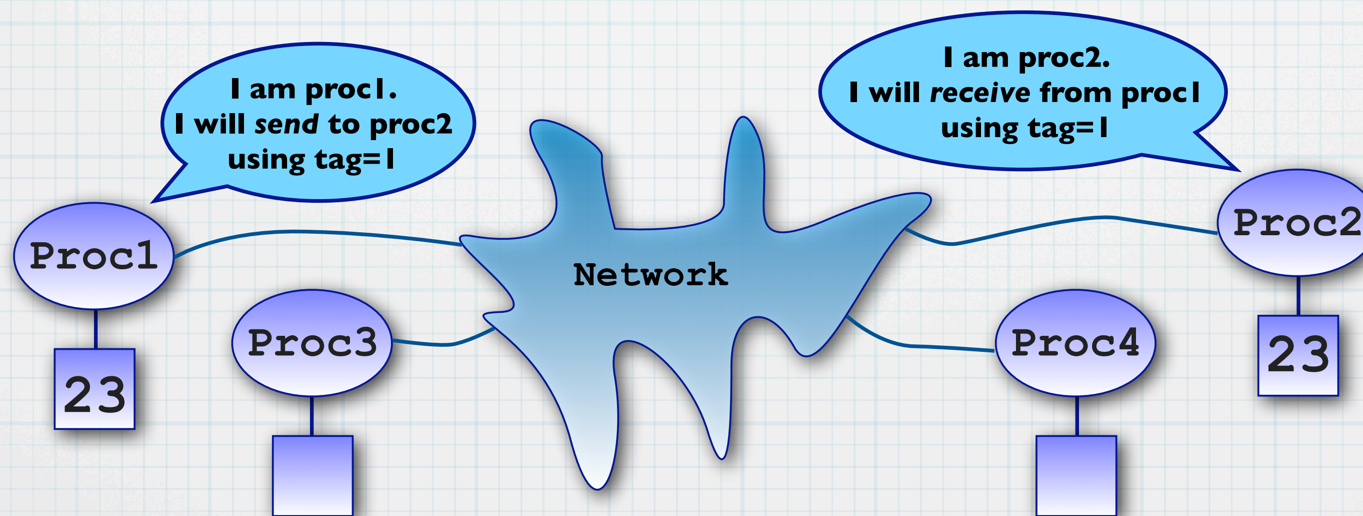
  http://www.mpi-forum.org/docs/mpi-11-html/node182.html

* What is MPI?

  1) MPI allows a collection of processes to communicate with messages.
  2) MPI is a library of subroutines called from Fortran, C and C++. Programs are compiler with ordinary compilers and linked with the MPI library.
  3) MPI is a specification which is independent from particular implementations. An MPI program should be portable to any vendors hardware that supports MPI.

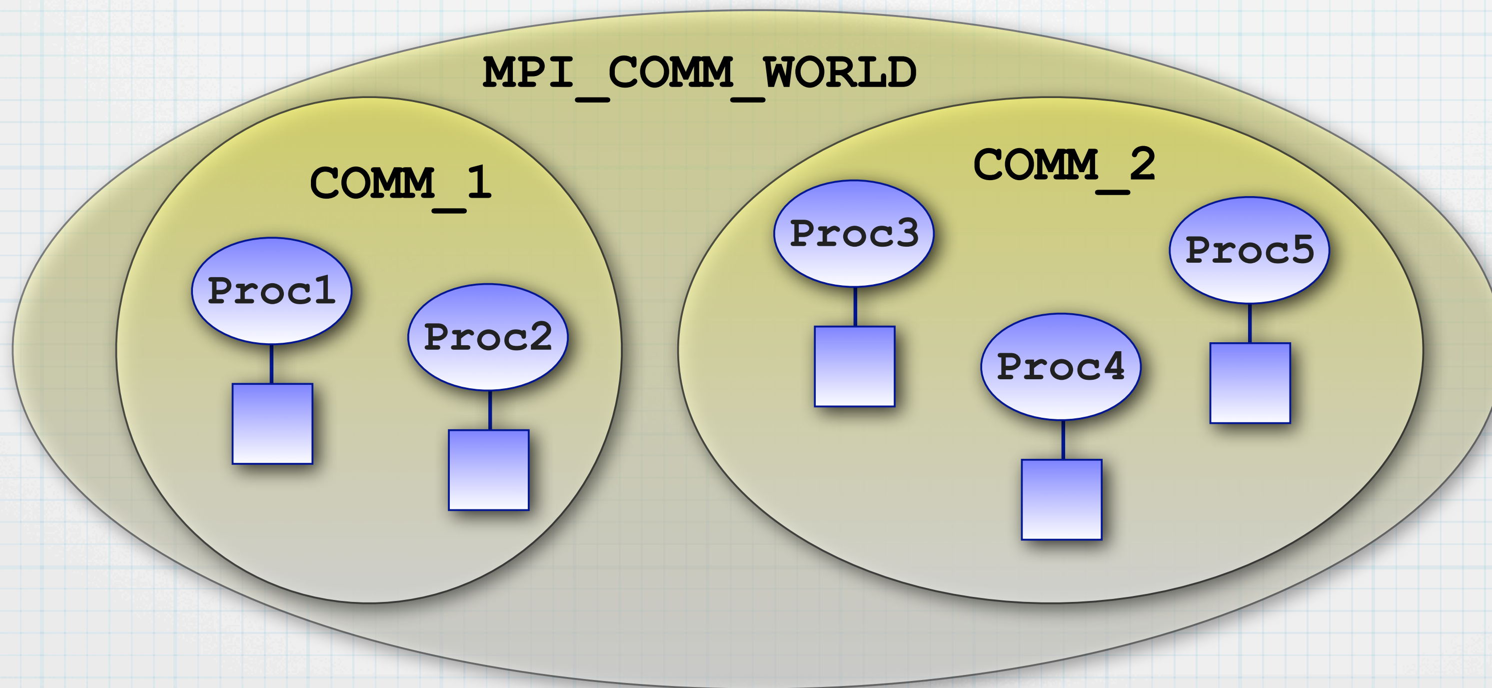
A minimal message-passing model

* The processes execute in parallel and have separate address spaces.
* Communication is cooperative. A message requires one process to execute a send command, and one process to execute a receive command.
* Information from one process's address space (memory) is transfered to another address space (memory) using a message.
* The two processes involved in the communication must agree upon a message tag to distinguish a message from other messages.



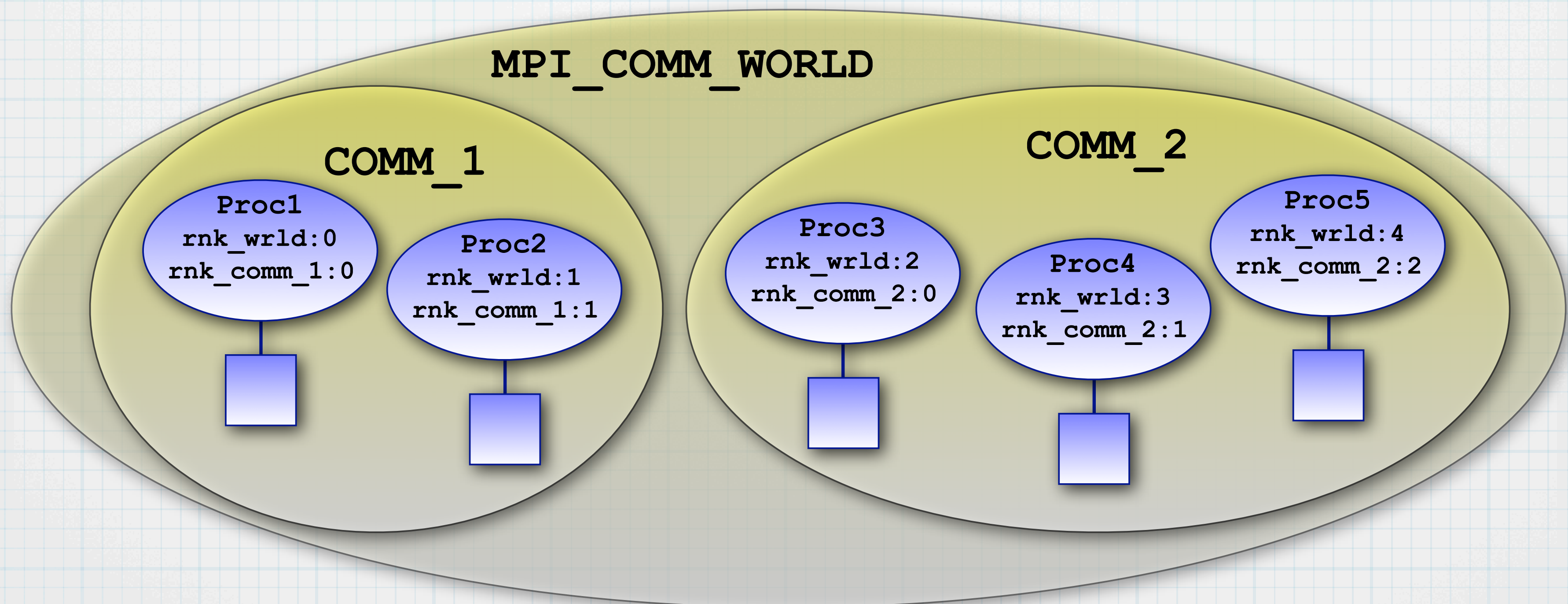
Communicators

* Groups of processes are called communicators.
  * The default communicator is called MPI_COMM_WORLD. This communicator contains all the processes in the current MPI universe.
  * MPI allows for the formation of communicators within the global communicator.
  * Message tags are defined within the context of a communicator.



Rank

- Processes are identified within a communicator by their rank
  - Rank is an integer
  - Rank defined within the context of a communicator.
  - If a communicator contains n processes, then the ranks are integers from 0 to n-1.



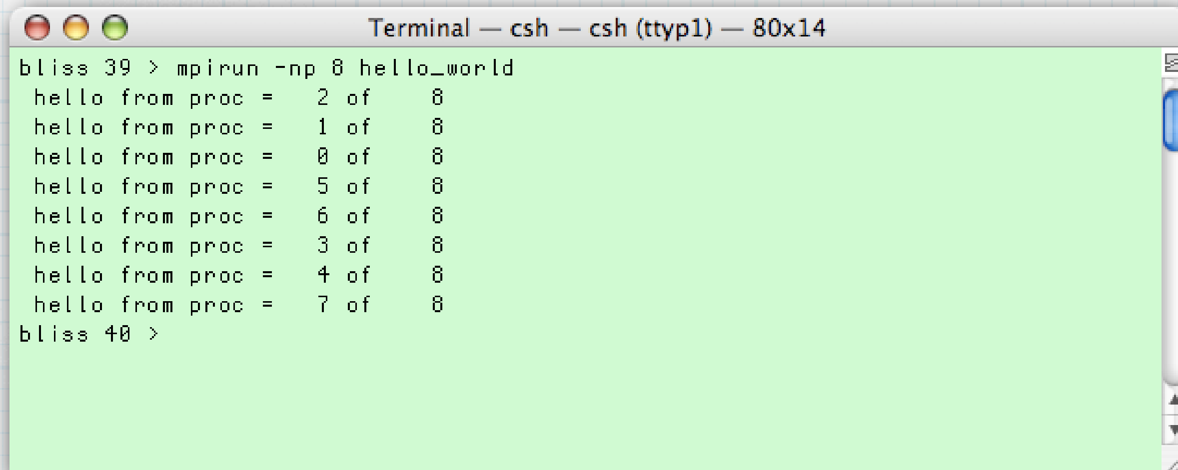
The "hello world" Program

- Important features of the hello_world program
  1. Use the `mpi` module, or include the include file called `mpif.h`
  2. Initialize the MPI environment.
  3. Determine how many processes are in the current MPI environment.
  4. Determine rank within the MPI_COMM_WORLD communicator
  5. Terminate the MPI environment

```
PROGRAM hello_world
USE mpi

IMPLICIT NONE
INTEGER :: npe_wrld, &! number of processes within the world communicator
           rnk_wrld, &! rank of process within the world communicator
           ierr

CALL MPI_INIT (ierr)                                ! initialize MPI environment
CALL MPI_COMM_SIZE (MPI_COMM_WORLD,npe_wrld,ierr) ! determine world size
CALL MPI_COMM_RANK (MPI_COMM_WORLD,rnk_wrld,ierr) ! determine rank within world

PRINT "(A19,I3,A4,I4)"," hello from proc = ",rnk_wrld," of ",npe_wrld

CALL MPI_FINALIZE (ierr)                            ! terminate MPI environment

END PROGRAM hello_world
```

Running the hello_world Program

```
PROGRAM hello_world
USE mpi

IMPLICIT NONE
INTEGER :: npe_wrld, &! number of processes within the world communicator
           rnk_wrld, &! rank of process within the world communicator
           ierr

CALL MPI_INIT (ierr)                                ! initialize MPI environment
CALL MPI_COMM_SIZE (MPI_COMM_WORLD,npe_wrld,ierr) ! determine world size
CALL MPI_COMM_RANK (MPI_COMM_WORLD,rnk_wrld,ierr) ! determine rank within world

PRINT "(A19,I3,A4,I4)"," hello from proc = ",rnk_wrld," of ",npe_wrld

CALL MPI_FINALIZE (ierr)                            ! terminate MPI environment

END PROGRAM hello_world
```



The slightly modified "hello world" Program

- Important features of the slightly modified hello_world program
  1. Use the mpi commands MPI_GET_PROCESSOR_NAME to determine where a processes is actually running.
  2. Use the mpi commands MPI_WTICK and MPI_WTIME to time code
  3. Use the mpi commands MPI_BARRIER write output in order.

```
PROGRAM hello_world_2
USE mpi
IMPLICIT NONE

INTEGER :: npe_wrld, &! number of processes within the world communicator
           rnk_wrld, &! rank of process within the world communicator
           i,j,n,name_len,ierr
REAL (KIND=SELECTED_REAL_KIND (12)) :: wall_tick,time_start,time_end,x

CHARACTER (LEN=128) :: proc_name

CALL MPI_INIT (ierr)
CALL MPI_COMM_SIZE (MPI_COMM_WORLD,npe_wrld,ierr)
CALL MPI_COMM_RANK (MPI_COMM_WORLD,rnk_wrld,ierr)
```

The slightly modified "hello world" Program

* Code (continued) for the slightly modified hello_world program

```
   CALL MPI_GET_PROCESSOR_NAME (proc_name,name_len,ierr)

   wall_tick  = MPI_WTICK () ! wall clock timer increment in seconds
   IF (rnk_wrld == 0)  PRINT "(A13,F12.8)"," wall_tick = ",wall_tick

! do some useless work
   time_start = MPI_WTIME () ! wall clock timer start
   x = 0.0_8
   DO j = 1,5000
      DO i = 1,5000
         x = x + SIN (x+FLOAT (rnk_wrld))
      ENDDO
   ENDDO
   time_end   = MPI_WTIME () ! wall clock timer stop

! write the results
   DO n = 0,npe_wrld-1
      IF (rnk_wrld == n) THEN
         PRINT "(A19,I3,A4,I4,A12,A16,A10,F8.5,A10,F12.8)", &
                                    " hello from proc = ",rnk_wrld," of ",npe_wrld, &
                                    " running on ",TRIM (proc_name), &
                                    " time = ",time_end-time_start," answer = ",x
      ENDIF
      CALL MPI_BARRIER (MPI_COMM_WORLD,ierr)
   ENDDO

   CALL MPI_FINALIZE (ierr)

   END PROGRAM hello_world_2
```

Collective Communication: Scatter

- Transfer information for one process to many (scatter) or collect information from many processes to one (gather)

- MPI_BCAST broadcasts a message from the process with rank ROOT to all processes of the communicator group COMM, itself included. It is called by all members of group using the same arguments. On return, the contents of root's send buffer has been copied to the receive buffer on all processes.

```
MPI_BCAST (buffer,data_count,data_type,root,comm)
```

Collective Communication: Gather

* `MPI_GATHER`: Each process (root process included) sends the contents of its send buffer to the `root` process. The root process receives the messages into the receive buffer and stores them in rank order.

```
MPI_GATHER(send_buffer,send_count,send_type,
  recv_buffer,recv_count,recv_type,root,comm,ierr)
```

* `MPI_REDUCE`: Combines the elements in the send buffer of each process in the communicator group `comm`, using the operation `op`, and returns the combined value in the receive buffer of the process with rank `root`.

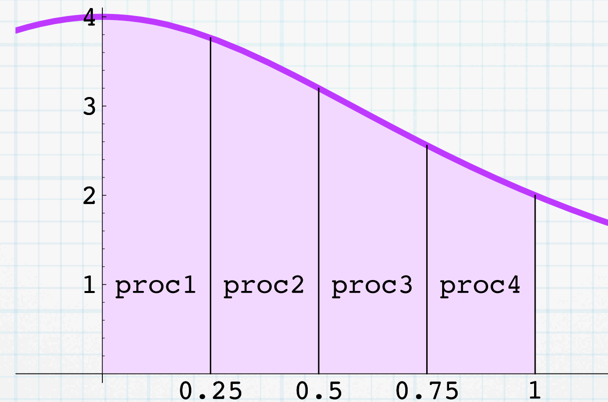
```
MPI_REDUCE(send_buffer,recv_buffer,send_count,
       send_type,op,root,comm,ierr)
```

where `op` can be several things including `MPI_MAX` (`MPI_MIN`) for maximum (minimum), `MPI_SUM` for summation.

An example with MPI_BCAST and MPI_REDUCE

* Find an approximation for π using numerical integration

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



* The algorithm for the code:

1. The root process will read the global number of intervals and broadcast the number to the other processes using MPI_BCAST.
2. Each process will then determine its subinterval using its rank in the communicator and integrate to find its subarea
3. Using MPI_REDUCE with the option MPI_SUM the subareas are summed to find the total area

Code the Pi example

```
   PROGRAM pi
   USE mpi
   IMPLICIT NONE

   INTEGER :: npe_wrld,rnk_wrld,n,i,ierr
   REAL (KIND=SELECTED_REAL_KIND (12)) :: &
      del_x,x_left,pi_piece,pi_approx,time_start,time_end,x
! setup MPI
   CALL MPI_INIT (ierr)
   CALL MPI_COMM_SIZE (MPI_COMM_WORLD,npe_wrld,ierr)
   CALL MPI_COMM_RANK (MPI_COMM_WORLD,rnk_wrld,ierr)
! read and broadcast total number of intervals
   IF (rnk_wrld==0) THEN
      PRINT *, 'Enter the total number of intervals '
      READ (*,*) n
   ENDIF
   CALL MPI_BCAST (n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

   time_start = MPI_WTIME () ! wall clock timer start
! integrate subinterval
   del_x = 1.0_8/DBLE (n); x_left = DBLE (rnk_wrld)/DBLE (npe_wrld);
   pi_piece = 0.0_8
   DO i = 1,n/npe_wrld
      x = x_left + del_x*(DBLE(i)-0.5_8)
      pi_piece = pi_piece + del_x*(4.0_8/(1.0_8 + x**2))
   ENDDO
! gather the pieces of the pi
   CALL MPI_REDUCE(pi_piece,pi_approx,1,MPI_DOUBLE_PRECISION,MPI_SUM,0, &
                                                       MPI_COMM_WORLD,ierr)
   time_end = MPI_WTIME () ! wall clock timer stop
! print the approximate value
   IF (rnk_wrld==0) THEN
      PRINT "(A12,F22.20)","pi_approx = ",pi_approx
   ENDIF
   CALL MPI_BARRIER (MPI_COMM_WORLD,ierr)
   PRINT "(A12,F14.10)","time      = ",time_end-time_start

   CALL MPI_FINALIZE (ierr)

   END PROGRAM pi
```

Point-to-point communication

* Here we send messages directly form one process to another.
* MPI_SEND:
  - This is a blocking send. Control does not return until the message data has been safely stored away so that the sender is free to overwrite the send buffer.
  - The syntax of the blocking send operation is given below:

```
MPI_SEND (BUFFER,DATA_COUNT,DATA_TYPE,DEST,TAG,
                            COMM,IERR)
```

where

DEST is the rank of destination (integer) within COMM

TAG is the message tag (integer)

Point-to-point communication

* MPI_RECV:
  - This is a blocking receive. Control returns only after the receive buffer contains the newly received message.
  - The syntax of the blocking send operation is given below:

```
MPI_RECV (BUFFER,DATA_COUNT,DATA_TYPE,SOUR,TAG,COMM,
                              STATUS,IERR)
```

where

SOUR is the rank of source (integer) within COMM. The source can also be specified as MPI_ANY_SOURCE

TAG is the message tag (integer). The tag can also be specified as MPI_ANY_TAG

Point-to-point communication. Matrix-vector multiplication.

- This is a "master-slave" algorithm. One process (the master) is responsible for the coordinating the work of the others (the slaves).
- We wish to perform a matrix-vector multiply in parallel.

$$\mathbf{A}b = c$$

- The master's algorithm for the code:
  1. The master will broadcast the vector b to all the slaves.
  2. The master will send one row of the matrix **A** to each slave.
  3. The master then waits for the slave to perform the dot product and return the element of c. At this time the master sends that slave a new row of **A**. Continue until all rows are processed.
- The slave's algorithm for the code:
  1. The slaves receive vector b from master.
  2. Perform dot-products of b and rows of **A**. Send result to master

Point-to-point communication. Matrix-vector multiplication.

* The code is clearly partitioned into a master part and a slave part

```
PROGRAM mat_vec
USE mpi
IMPLICIT NONE

INTEGER,PARAMETER :: rows=100,cols=100
INTEGER :: npe_wrld,rnk_wrld,master,i,j,count_rows,sender,row_index,ierr
INTEGER :: status(MPI_STATUS_SIZE)
REAL (KIND=SELECTED_REAL_KIND (12)) :: &
   a(rows,cols),b(cols),c(rows),buffer(cols),ans,time_start,time_end

CALL MPI_INIT (ierr)
CALL MPI_COMM_SIZE (MPI_COMM_WORLD,npe_wrld,ierr)
CALL MPI_COMM_RANK (MPI_COMM_WORLD,rnk_wrld,ierr)

master = 0

IF (rnk_wrld==master) THEN ! THE MASTER DOES THIS BLOCK OF CODE
   .
    .
     .
ELSE ! THE SLAVES DO THIS BLOCK OF CODE
   .
    .
     .
ENDIF

CALL MPI_FINALIZE (ierr)

END PROGRAM mat_vec
```

Point-to-point communication. Matrix-vector multiplication.

* The master code looks like this:

```
DO j = 1,cols ! make an arbitrary matrix a and vector b
   b(j) = 1.0_8
   DO i = 1,rows
      a(i,j) = DBLE (i+j)
   ENDDO
ENDDO
CALL MPI_BCAST (b,cols,MPI_DOUBLE_PRECISION,master,MPI_COMM_WORLD,ierr)

count_rows = 0
DO i = 1,npe_wrld-1
   DO j = 1,cols
      buffer(j) = a(i,j)
   ENDDO
   CALL MPI_SEND (buffer,cols,MPI_DOUBLE_PRECISION,i,i,MPI_COMM_WORLD,ierr)
   count_rows = count_rows+1
ENDDO

DO i = 1,rows
   CALL MPI_RECV (ans,1,MPI_DOUBLE_PRECISION, &
                        MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,status,ierr)
   sender    = status(MPI_SOURCE)
   row_index = status(MPI_TAG) ! tag value in status is the row index
   c(row_index) = ans
   IF (count_rows < rows) THEN ! more work to be done.  send another row
      DO j = 1,cols
         buffer(j) = a(count_rows+1,j)
      ENDDO
      CALL MPI_SEND (buffer,cols,MPI_DOUBLE_PRECISION, &
                               sender,count_rows+1,MPI_COMM_WORLD,ierr)
      count_rows = count_rows+1
   ELSE ! tell sender that there is no more work
      CALL MPI_SEND (MPI_BOTTOM,0,MPI_DOUBLE_PRECISION,sender,0,MPI_COMM_WORLD,ierr)
   ENDIF
ENDDO
```

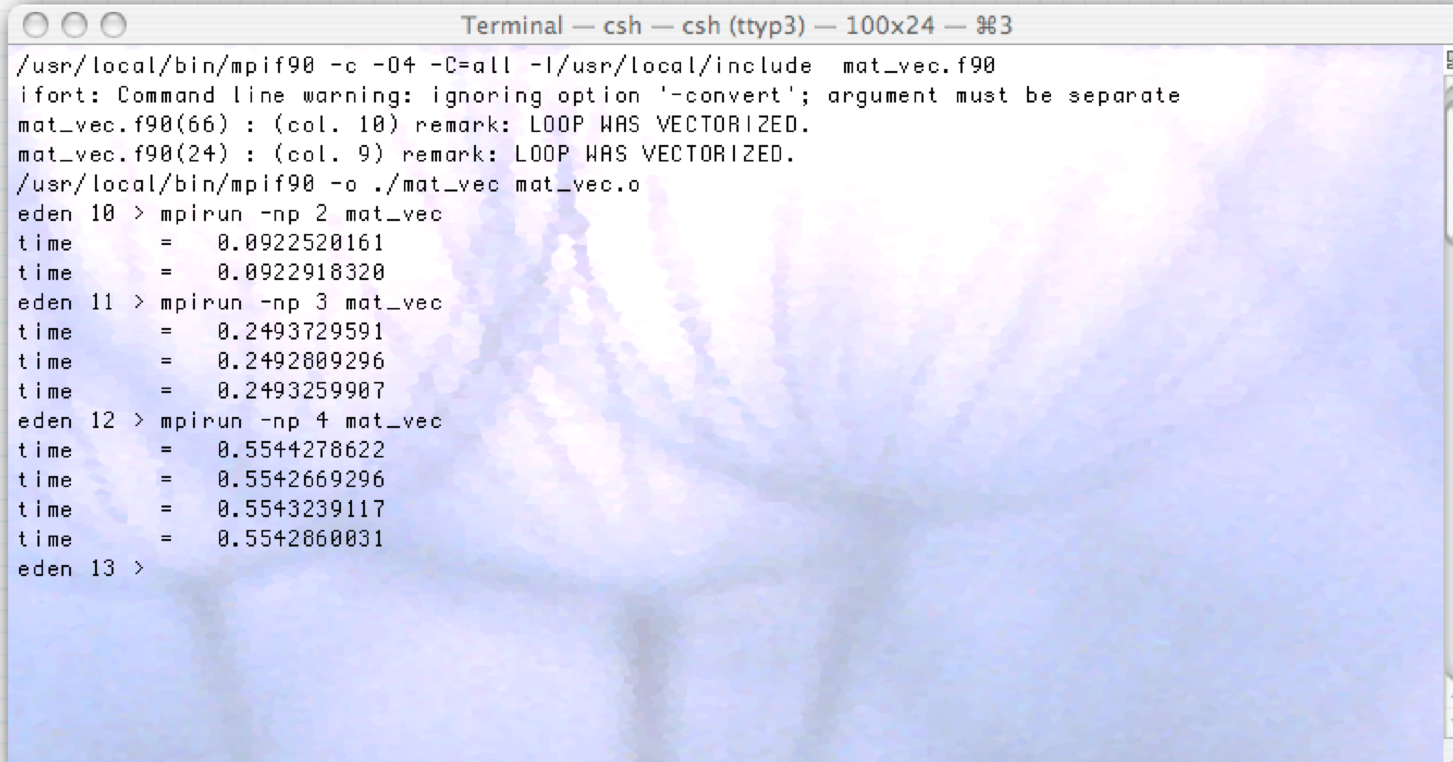
Point-to-point communication. Matrix-vector multiplication.

- The slave code looks like this:

```
CALL MPI_BCAST (b,cols,MPI_DOUBLE_PRECISION,master,MPI_COMM_WORLD,ierr)
DO
   CALL MPI_RECV (buffer,cols,MPI_DOUBLE_PRECISION,master, &
                                    MPI_ANY_TAG,MPI_COMM_WORLD,status,ierr)
   IF (status(MPI_TAG)==0) EXIT ! there is no more work
   row_index = status(MPI_TAG)  ! tag value status is the row index
   ans = 0.0_8
   DO i = 1,cols
      ans = ans + buffer(i)*b(i)
   ENDDO
   CALL MPI_SEND (ans,1,MPI_DOUBLE_PRECISION, &
                                      master,row_index,MPI_COMM_WORLD,ierr)
ENDDO
```

Point-to-point communication. Matrix-vector multiplication.

* Running the code
* slower with more processes...



Next time...

1. Non-blocking sends and receives
   - Overlapping communications and computations
2. Topologies
3. MPI datatypes