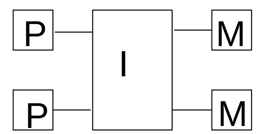
Lecture 2

Logical Abstractions of Multiprocessors

Physical Organization

- Uniform memory access (UMA) machines

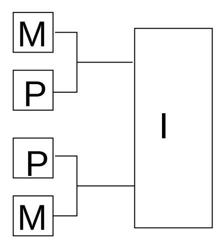


All memory is equally far away from all processors.

Early parallel processors like NYU Ultracomputer

Problem: why go across network for instructions? read-only data?
what about caches?

- Non-uniform memory access (NUMA) machines:



Access to local memory is usually 10-1000 times faster than access to non-local memory

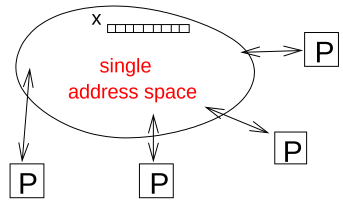
Static and dynamic locality of reference are critical for high performance.

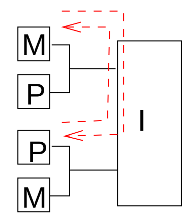
Compiler support? Architectural support?

Bus-based symmetric multiprocessors (SMP's): combine both aspects

Logical Organization

- Shared Memory Model

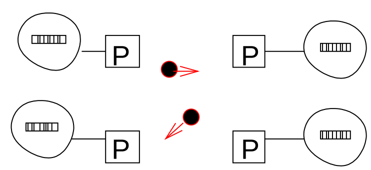


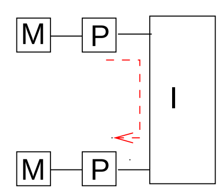


(conceptual picture)

- hardware/systems software provide single address space model to applications programmer
- some systems: distinguish between local and remote references
- communication between processors: read/write shared memory locations: put get

- Distributed Memory Model (Message Passing)





(conceptual picture)

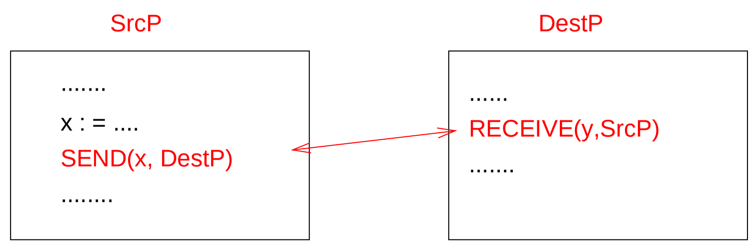
- each processor has its own address space
- communication between processors: messages (like e-mail)
- basic message-passing commands: send receive

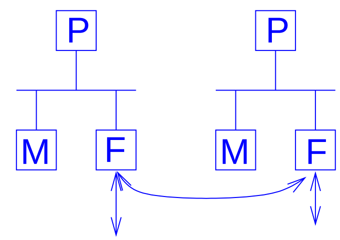
Key difference: In SMM, P1 can access remote memory locations w/o prearranged participation of application program on remote processor

Message Passing

Blocking SEND/RECEIVE : couple data transfer and synchronization

- Sender and receiver rendezvous to exchange data





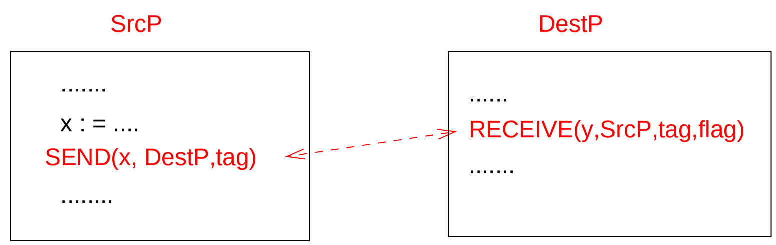
History: Caltech Cosmic Cube

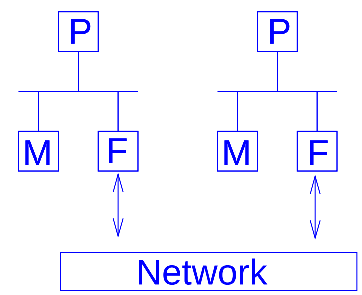
- SrcP field in RECEIVE command permits DestP to select which processor it wants to receive data from
- Implementation:
  - SrcP sends token saying ‘ready to send’
  - DestP returns token saying ‘me too’
  - Data transfer takes place directly between application programs w/o buffering in O/S
- Motivation: Hardware ‘channels’ between processors in early multicomputers
- Problem:
  - sender cannot push data out and move on
  - receiver cannot do other work if data is not available yet

    one possibility: new command TEST(SrcP,flag): is there a message from SrcP?

Overlapping of computation and communication is critical for performance

Non-blocking SEND/RECEIVE : decouple synchronization from data transfer



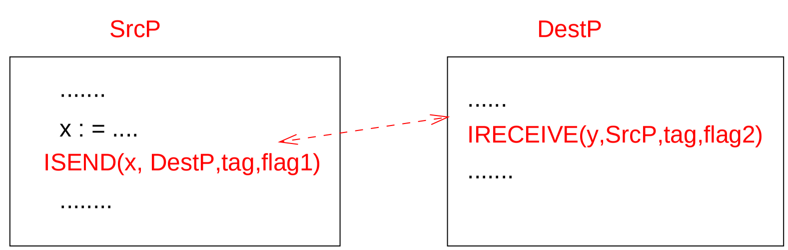


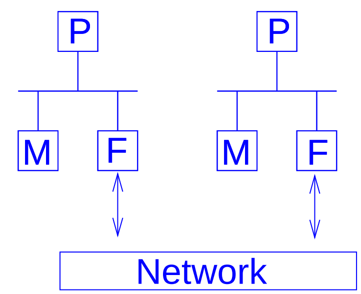
- SrcP can push data out and move on
- Many variation: return to application program when
  - data is out on network?
  - data has been copied into an O/S buffer?
- Tag field on messages permits receiver to receive messages in an order different from order that they were sent by SrcP
- RECEIVE does not block
  - flag is set to true by O/S if data was transfered/false otherwise
- Applications program can test flag and take the right action
- What if DestP has not done a RECEIVE when data arrives from SrcP?
- Data is buffered in O/S buffers at DestP till application program does a RECEIVE

Can we eliminate waiting at SrcP ?

Can we eliminate buffering of data at DestP ?

Asynchronous SEND/RECEIVE





- SEND returns as soon as O/S knows about what needs to be sent
- ‘Flag1’ set by O/S when data in x has been shipped out
- Application program continues, but must test ‘flag1’ before overwriting x
- RECEIVE is non-blocking:
  - returns before data arrives
  - tells O/S to place data in ‘y’ and set ‘flag’ after data is received
  - ‘posting’ of information to O/S
  - ‘Flag2’ is written by O/S and read by application program on DestP
- Eliminates buffering of data in DestP O/S area if IRECEIVE is posted before message arrives at DestP

So far, we have looked at point-to-point communication

Collective communication:

- patterns of group communication that can be implemented more efficiently than through long sequences of send's and receive's

- important ones:

  - one-to-all broadcast

    (eg. A*x implemented by rowwise distribution: all processors need x)

  - all-to-one reduction

    (eg. adding a set of numbers distributed across all processors)

  - all-to-all broadcast

    every processor sends a piece of data to every other processor

  - one-to-all personalized communication

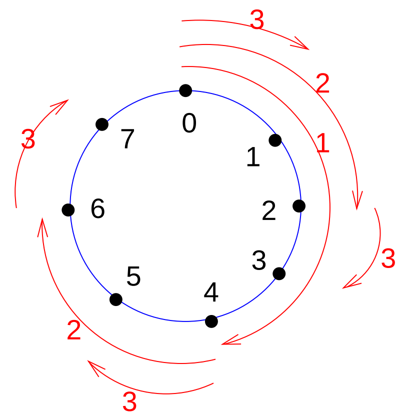
    one processor sends a different piece of data to all other processors

  - all-to-all personalized communication

    each processor does a one-to-all communication

Example: One-to-all broadcast

(intuition: think 'tree')



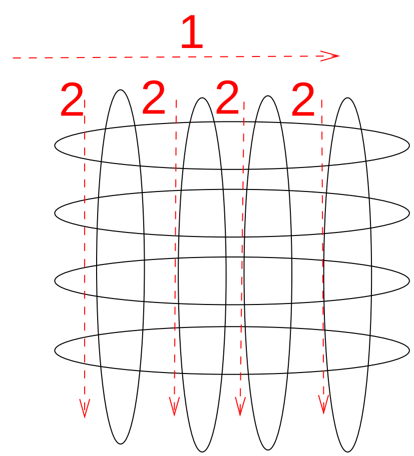
Messages in each phase
do not compete for links

Assuming message size is small, time to send a message = Ts + h*Th
where Ts = overhead at sender/receiver
Th = time per hop

Total time for broadcast = Ts + Th*P/2
+ Ts + Th*P/4
+ .....
= Ts * logP + Th*(P-1)

Reality check: Actually, a k-ary tree makes sense because processor 0 can send many messages by the time processor 4 is ready to participate in broadcast

Other topologies: use the same idea



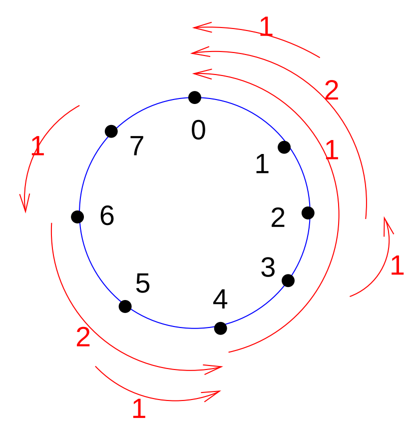
2-D Mesh

Step 1: Broadcast within row of originating processor

Step 2: Broadcast within each column in parallel

Time = Ts logP + 2Th*(sqrt(P) -1)

Example: All-to-one reduction



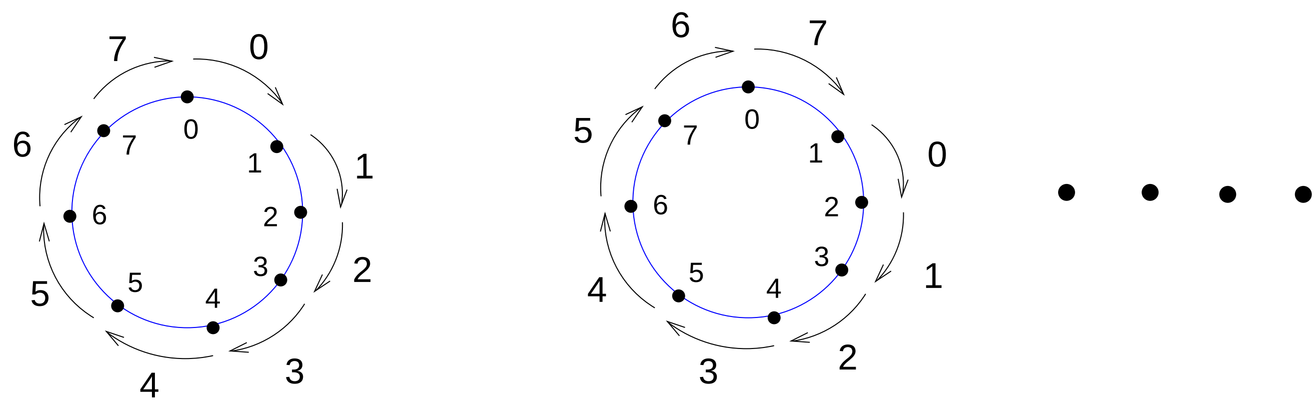
Messages in each phase
do not compete for links

Purpose: apply a commutative and associative operator (reduction operator) like +,*,AND,OR etc to values contained in each node

Can be viewed as inverse of one-to-all broadcast
Same time as one-to-all broadcast

Important use: determine when all processors are finished working (implementation of ‘barrier’)

Example: All-to-all broadcast



- Intuition: cyclic shift register
- Each processor receives a value from one neighbor , stores it away, and sends it to next neighbor in the next phase.
- Total of (P-1) phases to complete all-to-all broadcast

Time = (Ts + Th) *(P-1) assuming message size is small

- Same idea can be applied to meshes as well:
  - first phase, all-to-all broadcast within each row
  - second phase, all-to-all broadcast within each column

A Message-passing Program

MPI: Message-Passing Interface

Goal: Portable Parallel Programming for Distributed Memory Computers

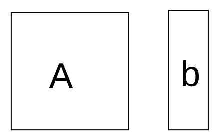
- Lots of vendors of Distributed Memory Computers:
  IBM,NCube, Intel, CM-5, ...

- Each vendor had its own communication constructs
  => porting programs required changing parallel programs
  even to go from one distributed memory platform to another!

- MPI goal: standardize message passing constructs syntax and semantics

- Mid 1994: MPI-1 standard out and several implementations available (SP-2)

Write an MPI program to perform matrix-vector multiply



- Style of programming: Master-Slave
  - one master, several slaves
  - master co-ordinates activities of slaves
- Master initially owns all rows of A and vector b
- Master broadcasts vector b to all slaves
- Slaves are self-scheduled
  - each slave comes to master for work
  - master sends a row of matrix to slave
  - slave performs product, returns result and asks for more work
- Very naive algorithm, but it's a start.

Key MPI Routines we will use:

MPI_INIT : Initialize the MPI System
MPI_COMM_SIZE: Find out how many processes there are
MPI_COMM_RANK: Who am I?
MPI_SEND: Send a message

MPI_SEND(address,count,datatype,DestP,tag,comm)

permits entire data structures to be sent with one command

identifies process group

MPI_RECV: Receive a message (blocking receive)
MPI_FINALIZE: Terminate MPI
MPI_BCAST: Broadcast

```fortran
c   COMMON PROGRAM EXECUTED BY BOTH MASTER AND SLAVES
c************************************************************************
c     matmul.f - matrix - vector multiply, simple self-scheduling version
c**************************************************************************
      program main
      include 'mpif.h'
      integer MAX_ROWS, MAX_COLS, rows, cols
      parameter (MAX_ROWS = 1000, MAX_COLS = 1000)
      double precision a(MAX_ROWS,MAX_COLS), b(MAX_COLS), c(MAX_COLS)
      double precision buffer(MAX_COLS), ans
      integer myid, master, numprocs, ierr, status(MPI_STATUS_SIZE)
      integer i, j, numsent, numrcvd, sender, job(MAX_ROWS)
      integer rowtype, anstype, donetype

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

      master   = 0
      rows     = 100
      cols     = 100

      if ( myid .eq. master ) then
c        master initializes and then dispatches
         .......
      else
c        slaves receive b, then compute dot products until done message
         .......
      endif

 200  call MPI_FINALIZE(ierr)
      stop
      end
```

```
c CODE EXECUTED BY MASTER
      if ( myid .eq. master ) then
c         master initializes and then dispatches
c         initialize a and b to arbitrary values
         do 20 i = 1,cols
            b(i) = 1
            do 10 j = 1,rows
               a(i,j) = i
 10         continue
 20      continue

         numsent = 0
         numrcvd = 0

c         send b to each other process
         call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
     $        MPI_COMM_WORLD, ierr)

c         send a row to each other process
         do 40 i = 1,numprocs-1
            do 30 j = 1,cols
               buffer(j) = a(i,j)
 30         continue

            call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, i,
     $           rowtype, MPI_COMM_WORLD, ierr)

c  Job(i) = NUMBER OF ROW CURRENTLY BEING PROCESSED BY PROCESS i.
            job(i)  = i
            numsent = numsent+1
 40      continue

         do 70 i = 1,rows
```

```
         call MPI_RECV(ans, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,
     $        anstype, MPI_COMM_WORLD, status, ierr)
         sender = status(MPI_SOURCE)
         c(job(sender)) = ans

         if (numsent .lt. rows) then
            do 50 j = 1,cols
               buffer(j) = a(numsent+1,j)
 50         continue
            call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, sender,
     $           rowtype, MPI_COMM_WORLD, ierr)
            job(sender) = numsent+1
            numsent     = numsent+1
         else
         call MPI_SEND(1, 1, MPI_INTEGER, sender, donetype,
     $        MPI_COMM_WORLD, ierr)
         endif
 70   continue

c     print out the answer
      do 80 i = 1,cols
         print *, "c(", i, ") = ", c(i)
 80      continue
```

```
c CODE EXECUTED BY SLAVES

c        slaves receive b, then compute dot products until done message
         call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
     $        MPI_COMM_WORLD, ierr)
 90      call MPI_RECV(buffer, cols, MPI_DOUBLE_PRECISION, master,
     $        MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
         if (status(MPI_TAG) .eq. donetype) then
            go to 200
         else
            ans = 0.0
            do 100 i = 1,cols
               ans = ans+buffer(i)*b(i)
 100        continue
            call MPI_SEND(ans, 1, MPI_DOUBLE_PRECISION, master, anstype,
     $           MPI_COMM_WORLD, ierr)
            go to 90
         endif
      endif

 200  call MPI_FINALIZE(ierr)
      stop
      end
```

```
c*********************************************************************
c     matmul.f - matrix - vector multiply, simple self-scheduling version
c***********************************************************************
      program main

      include 'mpif.h'

      integer MAX_ROWS, MAX_COLS, rows, cols
      parameter (MAX_ROWS = 1000, MAX_COLS = 1000)
      double precision a(MAX_ROWS,MAX_COLS), b(MAX_COLS), c(MAX_COLS)
      double precision buffer(MAX_COLS), ans

      integer myid, master, numprocs, ierr, status(MPI_STATUS_SIZE)
      integer i, j, numsent, numrcvd, sender, job(MAX_ROWS)
      integer rowtype, anstype, donetype

      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
      if (numprocs .lt. 2) then
         print *, "Must have at least 2 processes!"
         MPI_Abort( MPI_COMM_WORLD, 1 )
         stop
      endif
      print *, "Process ", myid, " of ", numprocs, " is alive"

      rowtype  = 1
      anstype  = 2
      donetype = 3

      master   = 0
      rows     = 100
      cols     = 100
```

```
      if ( myid .eq. master ) then
c        master initializes and then dispatches
c        initialize a and b
         do 20 i = 1,cols
            b(i) = 1
            do 10 j = 1,rows
               a(i,j) = i
 10         continue
 20      continue

         numsent = 0
         numrcvd = 0

c        send b to each other process
         call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
     $        MPI_COMM_WORLD, ierr)

c        send a row to each other process
         do 40 i = 1,numprocs-1
            do 30 j = 1,cols
               buffer(j) = a(i,j)
 30         continue
            call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, i,
     $           rowtype, MPI_COMM_WORLD, ierr)
            job(i)  = i
            numsent = numsent+1
 40      continue

         do 70 i = 1,rows
            call MPI_RECV(ans, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,
     $           anstype, MPI_COMM_WORLD, status, ierr)
            sender = status(MPI_SOURCE)
```

```
            c(job(sender)) = ans

            if (numsent .lt. rows) then
               do 50 j = 1,cols
                  buffer(j) = a(numsent+1,j)
 50            continue
               call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, sender,
     $              rowtype, MPI_COMM_WORLD, ierr)
               job(sender) = numsent+1
               numsent     = numsent+1
            else
            call MPI_SEND(1, 1, MPI_INTEGER, sender, donetype,
     $           MPI_COMM_WORLD, ierr)
            endif
 70      continue

c        print out the answer
         do 80 i = 1,cols
            print *, "c(", i, ") = ", c(i)
 80         continue

      else
c        slaves receive b, then compute dot products until done message
         call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
     $        MPI_COMM_WORLD, ierr)
 90      call MPI_RECV(buffer, cols, MPI_DOUBLE_PRECISION, master,
     $        MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
         if (status(MPI_TAG) .eq. donetype) then
            go to 200
         else
            ans = 0.0
            do 100 i = 1,cols
               ans = ans+buffer(i)*b(i)
```

```
100       continue
          call MPI_SEND(ans, 1, MPI_DOUBLE_PRECISION, master, anstype,
     $         MPI_COMM_WORLD, ierr)
          go to 90
        endif
      endif

200   call MPI_FINALIZE(ierr)
      stop
      end
```

This style of parallel programming is called

Single Program Multiple Data (SPMD) programming

All processors execute the same code but branch on their IDs to perform disjoint activities.

In principle, each processor could run different programs, but this is not very common (cf. CSP, OCCAM, ...).