Logical Abstractions of Multiprocessors
Bus-based symmetric multiprocessors (SMP's): Combine both aspects

Compiler support? Architectural support?

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

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

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

Early parallel processors like NYU Ultracomputer

All memory is equally far away from all processors.

Problem: Why go across network for instructions? Read-only data?

Non-uniform memory access (NUMA) machines:

What about caches?

Uniform memory access (UMA) machines:

Physical Organization
**Logical Organization**

**Shared Memory Model**
- Hardware/software provide single address space

**Distributed Memory Model (Message Passing)**
- Communication between processors: read/write shared memory locations
  - Some systems distinguish between local and remote references
  - In applications programmer
- Communication between processors: messages (like e-mail)

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

Basic message-passing commands:
- send, receive
- put, get

References
Message Passing
Sender and receiver rendezvous to exchange data

Overlapping of computation and communication is critical for performance.

one possibility: new command TEST(SrcP,flag): is there a message from SrcP?
- receiver cannot do other work if data is not available yet
- sender cannot push data out and move on

Problem:

Motivation: Hardware channels' between processors in early multi-computers
w/o buffering in O/S
- Data transfer takes place directly between application programs
- DesP returns token saying 'me too'
- SrcP sends token saying 'ready to send'

Implementation:

which processor it wants to receive data from

SrcP field in RECEIVE command permits DesP to select

Implementation:

History: Caltech Cosmic Cube

Sender and receiver rendezvous to exchange data

Blocking SEND/RECEIVE: couple data transfer and synchronization
Non-blocking SEND/RECEIVE

deounce synchronization from data transfer

Can we eliminate waiting at DestP?

Data is buffered in O/S buffers at DestP till application program does a RECEIVE

What if DestP has not done a RECEIVE when data arrives from SrcP?

Application program can test flag and take the right action

- flag is set to true by O/S if data was transferred/false otherwise
- RECEIVE does not block
- in an order different from order they were sent by SrcP
- Tag field on messages permits receiver to receive messages
- data has been copied into an O/S buffer?
- data is on O/S network?

- Many variation: return to application program when
- DestP can push data out and move on

SrcPDestP

SEND(x, DestP, tag)

RECEIVE(y, SrcP, tag, flag)

- 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?
Before message arrives at DestP:

- Eliminates buffering of data in DestP's area if IRECEIVE is posted.
- 'Flag2' is written by O/S and read by application program on DestP.
- Posting of information to O/S

Tells O/S to place data in 'y' and set 'Flag' after data is received.

Returns before data arrives.

- RECEIVE is non-blocking:
  - Application program continues, but must test 'Flag' before overwriting x.
- 'Flag1' set by O/S when data in x has been shipped out.
- SEND returns as soon as O/S knows about what needs to be sent.

Asynchronous SEND/RECEIVE
So far, we have looked at point-to-point communication.

Collective communication:

- point-to-point communication
- collective communication
  - patterns of group communication that can be implemented more efficiently
  - important ones:
    - point-to-point broadcast
    - all-to-all personalized communication
      - one processor sends a different piece of data to all other processors
    - all-to-all personalized communication
      - every processor sends a piece of data to every other processor
      - one-to-all broadcast
        - each processor sends a piece of data to every other processor
        - (eg. adding a set of numbers distributed across all processors)
      - all-to-one reduction
        - (eg. A \times x implemented by rowwise distribution: all processors need x)
    - one-to-all broadcast

- important ones:
  - then through long sequences of sends and receives
in broadcast

many messages by the time processor 4 is ready to participate

Reality check: Actually, a k-ary tree makes sense because processor 0 can send

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Example: One-to-all broadcast

Assuming message size is small, time to send a message = T_s + T_h

Reality check: Actually, a k-ary tree makes sense because processor 0 can send

\[
T_s \cdot \log_2 P = T_s + \frac{1}{2} T_h \cdot P
\]

\[
T_s + \frac{1}{4} T_h \cdot P
\]

Total time for broadcast = \( T_s + \frac{1}{2} T_h \cdot P \)

T_h = Time per hop

where \( T_s \) = overhead at sender/receiver

Messages in each phase (intuition: think tree)

Messages in each phase (intuition: think tree)

Messages in each phase
Step 1: Broadcast within row of originating processor

Step 2: Broadcast within each column in parallel

\[ T = T_s \log P + 2T_h \sqrt{P} - 1 \]

Other topologies: use the same idea
Messages in each phase do not compete for links.

Example: All-to-one reduction

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

Important use: determine when all processors are finished working.

Same time as one-to-all broadcast

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

Implementation of `barrier`

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.
- Second phase, all-to-all broadcast within each column
- First phase, all-to-all broadcast within each row
  - Same idea can be applied to meshes as well:
    - Time = (Ts + Th) * (P-1) assuming message size is small

  - Total of (P-1) phases to complete all-to-all broadcast

    - Each processor receives a value from one neighbor
    - Stores it away, and sends it to next neighbor in the next phase.
    - Initialization: cyclic shift register

Instance: All-to-all broadcast

Example:
A Message-Passing Program
Goal: Portable Parallel Programming for Distributed Memory Computers

- Each vendor had its own communication constructs

- MPI: Message-Passing Interface

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

- MPI goal: standardize message passing constructs syntax and semantics

- Even to go from one distributed memory platform to another!

- MPI: Message-Passing Interface

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

Distributed Memory Computers

Goal: Portable Parallel Programming for Distributed Memory Computers

- MPI: Message-Passing Interface
A master-slave style of programming:

- Slave performs product, returns result and asks for more work
- Master sends a row of matrix to slave
- Each slave comes to master for work
- Slaves are self-scheduled

Master broadcasts vector b to all slaves

- Master initially owns all rows of A and vector b
- Master co-ordinates activities of slaves
- One master, several slaves

- Style of programming: Master-Slave

Write an MPI program to perform matrix-vector multiply
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_BCAST: Broadcast
MPI_Finalize: Terminate MPI
MPI_RECV: Receive a message (blocking receive)

MPI_SEND(address, count, datatype, DestP, tag, comm)
permits entire data structures to be sent with one command
identifies process group

MPI_Finalize
MPI_RECV
MPI_BCAST
MPI_SEND(address, count, datatype, DestP, tag, comm)
end
stop

200 call MPI_FINALIZE( ierr )

end

......

else

......

master initializes and then dispatches

if ( mpi_id = 0 ) then

  cost = 100
  rows = 100
  master = 0

  call MPI_COMM_SIZE( MPI_COMM_WORLD, num_procs, ierr )
  call MPI_COMM_RANK( MPI_COMM_WORLD, mpi_id, ierr )
  call MPI_INIT( ierr )

  integer rows, columns, done_flag, double precision buffer( MAX-ROWS, MAX-COLS )
  integer mpi_id, num_procs, status, job( MAX-ROWS, MAX-COLS ), MX-ROWS, MX-COLS
  integer ( parameter ) MAX-ROWS = 1000, MAX-COLS = 1000
  integer MAX-ROWS, MAX-COLS, rows, columns
  integer mpi_id, rank, master, slaves, slaves received, done_flag, done_flag

program main

****************************************************************************
common program executed by both master and slaves

*****************************************************************************
do 10 j = 1, jmax

continue
numsent = numsent + 1
if (j) /= 1, i = 1, jmax
        call mpi_send(buffer, count, mpi_double-precision, i, mpi-comm-quad, ierr)
        call mpi_send(buffer, count, mpi_double-precision, i, mpi-comm-quad, ierr)
        continue
send a row to each other process
0 = numsent
0 = numsent
continue
continue
10
a(j, j) = 1
continue
10
b(j) = 1
continue
20
do 10 j = 1, jmax
continue
a(j, j) = 1
continue
do 20 j = 1, jmax
continue
if (myid == master) then
        code executed by master
continue
print("", c(t), ",")
do 80 i = 1, cols
print out the answer
continue
end
$\text\{MPI-\text{COMM-WORLD, }}\text{iter}\text{\}}$
call MPI\text{-}SEND(1, 1, \text{MPI\text{-}INTEGER, sender, done})
else
num sent = num sent + 1
job\{sender\} = num sent + 1
$\text{repeat, MPI-\text{COMM-WORLD, }iter\}}$
call MPI\text{-}SEND\{buffer, cols, MPI\text{-}DOUBLE\text{-}PRECISION, sender, }
continue
buffer\{f\} = (num sent + 1, f)
do 50 j = 1, cols
if (num sent \geq 1, f) then
$\text{job\{sender\}} = \text{null}$
$\text{sender} = \text{status}(\text{MPI\text{-}SOURCE,}$
$\text{MPI-\text{COMM-WORLD, status, iter\}}}$
call MPI\text{-}RECV\{ans, 1, MPI\text{-}DOUBLE\text{-}PRECISION, MPI\text{-}ANY\text{-}SOURCE, }
end
stop
200 call MPI_FINALIZE(ierr)
end
end
end

go to 90
$MPI.COMM_WORLD, ierr,
call MPI_SEND(ans, 1, MPI_DOUBLE_PRECISION, master, anstyp, 100)
continue
ans = ans+anxtyp(t)((t)
do 100 i = 1, 100
ans = 0.0
continue
else
if status(MPI_TAG, erro, done>type) then
MPI_ANY_TAG, MPI.COMM_WORLD, status, ierr,
call MPI_RECV(butter, 1, MPI_DOUBLE_PRECISION, master,
90 MPI.COMM_WORLD, ierr,
call MPI_BCAST(butter, 1, MPI_DOUBLE_PRECISION, master,
0, slaves receives b) then compute dot products until done message
$C CODE EXECUTED BY SLAVES
program main

**************************************************************************
c - matrix - vector multiply, simple self-scheduling version
**************************************************************************

cols = 100
rows = 100
master = 0

donepe = 3
mastepe = 2
rowope = 1

print * , 'process', mpid, 0 'numprocs' 'is active'
endt
stop

mpi-abort (mpi-com-world, 1)

if (numprocs .eq. 1) then
  call mpi-comm-size (mpi-com-world, numprocs, ierr)
  call mpi-comm-ranked (mpi-com-world, mpid, ierr)
end if

integer *8 mastepe, donepe, rowope
integer *2, j, numSent, numrcvd, job(MAX-HOWS)
integer *4, mpid, master, numprocs, ierr, status(MPI-STATS-SIZE)

double precision buffer(MAX-COLS), ans

double precision p(MAX-HOWS, MAX-COLS), d(MAX-COLS)

parameter (MAX-HOWS = 1000, MAX-COLS = 1000)

integer MAX-HOWS, MAX-COLS, rows, cols

include 'mpif'
sender = status(MPI_SOURCE)
  
  MPI_Type, MPI_SEND, status, ierr
  
call MPI_RECV (msg, i, MPI_DOUBLE PRECISION, MPI_ANY SOURCE, 
  
do to 1 = 1, ierr

  continue

  nonsent = nonsent+1
  
  job(1) = 1
  
call MPI_SEND (butter, cols, MPI_DOUBLE PRECISION, 

  continue

  butter(j) = a(j, j)
  
do to 1 = 1, cols
  do to 1 = 1, nonsent-1

  send a row to each other process

  MPI_SEND (b, cols, MPI_DOUBLE PRECISION, MASTER, 

  MPI_DOUBLE PRECISION, MASTER)

  send b to each other process

  numsent = 0
  0 = nonsent

  continue

  continue

  a(j, j) = 1
  
do to 1 = 1, cols
  do to 1 = 1, cols

  initialize a and b

  master initializes and then dispatches

  if (MPID == master) then
ans = ans + buffer(1) * (i - 1)
do 100 i = 1, core
   ans = 0.0
  else
    go to 200
  if status(MPI_TAG, '9', donetype) then
      MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr
      call MPI_RECV(buffer, core, MPI_DOUBLE_PRECISION, master, MPI_COMM_WORLD, ierr)
call MPI_BCAST(buffer, core, MPI_DOUBLE_PRECISION, master)
      c shares receive b, then compute dot products until done message
      continue
  endt
  print *, 'c''(i) = ' (c(i), i = 1, core)
do 80 i = 1, core
    print out the answer
  end
  continue
endt
  call MPI_SEND(buffer, core, MPI_DOUBLE_PRECISION, master, MPI_COMM_WORLD, ierr)
call MPI_SEND(buffer, core, MPI_DOUBLE_PRECISION, master, MPI_COMM_WORLD, ierr)
  continue
  do 50 j = 1, core
    continue
  endt
  if (numsend+1, j) > 0 then
    c[j](sender) ans = ans
end
stop

200 call mpi_finalize
end
end
end
go to 90

$mpi COMM WORLD, ierr)
call mpi_send (ans, 1, mpi_double precision, master, ansype,
continue
100
is not very common (e.g., CSP, OCaml, ...).

In principle, each processor could run different programs, but this
perform disjoint activities.

All processors execute the same code but branch on their IDs to
Single Program Multiple Data (SPMD) programming

This style of parallel programming is called