Week 3 – Introduction to MPI, parallel program design

- Main topics this week
  - Basics of MPI
  - Parallel Program Development
- Reading
  - Pacheco, Chapters 3 and 4

Lecture 4

Parallel architectures, MPI programming

Flynn's Taxonomy

- Instruction stream
- Data stream
- Single vs. multiple
- Four combinations
  - SISD
  - SIMD
  - MISD
  - MIMD

SISD

- Single Instruction, Single Data
- Single-CPU systems
- Note: co-processors don't count
  - Functional
  - I/O
- Example: PCs

SIMD

- Single Instruction, Multiple Data
- Two architectures fit this category
  - Pipelined vector processor (e.g., Cray-1)
  - Processor array (e.g., Connection Machine)

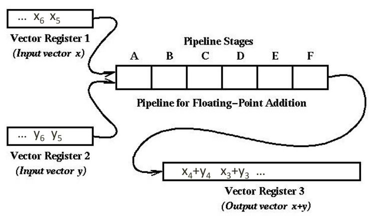
Where is vector computing found?

Old style: Cray computers

Rebirth: GPUs (graphics processing unit)
= specialized processor for 3D graphics rendering

Cell processor
= RISC core with coprocessing elements to accelerate multimedia and vector applications, in Playstation 3, etc.

Vector Computing



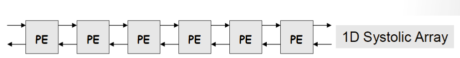
Why is this SIMD?

Toward a SIMD example: The stages of a floating-point addition pipe

- **Stage A:** The exponents of the two floating-point numbers to be added are compared to find the number with the smallest magnitude.
- **Stage B:** The significand of the number with the smaller magnitude is shifted so that the exponents of the two numbers agree.
- **Stage C:** The significands are added.
- **Stage D:** The result of the addition is normalized.
- **Stage E:** Checks are made to see if any floating-point exceptions occurred during the addition, such as overflow.
- **Stage F:** Rounding occurs.

MISD

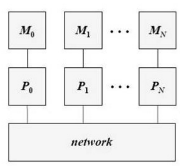
- Multiple Instruction, Single Data
- Example: systolic array



MIMD

- Multiple Instruction, Multiple Data
- Multiple-CPU computers
  - Multiprocessors
  - Multicomputers

Why is our simple distributed-memory machine MIMD?



distributed-memory multicomputer

SPMD

- SPMD (single program, multiple data): all processors execute same program, but each operates on different portion of problem data
- Easier to program than true MIMD and more flexible than SIMD
- Although most parallel computers today are MIMD architecturally, they are usually programmed in SPMD style

Thinking about distributed memory

- Algorithmically:
  - Topology of network affects algorithm design, implementation, and performance
  - Access to remote data requires communication

Switch Network Topologies

- View switched network as a graph
  - Vertices = processors or switches
  - Edges = communication paths
- Two kinds of topologies
  - Direct
  - Indirect

Direct Topology

- Ratio of switch nodes to processor nodes is 1:1
- Every switch node is connected to
  - 1 processor node
  - At least 1 other switch node

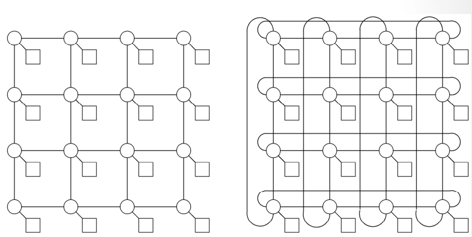
Indirect Topology

- Ratio of switch nodes to processor nodes is greater than 1:1
- Some switches simply connect other switches

Evaluating Switch Topologies

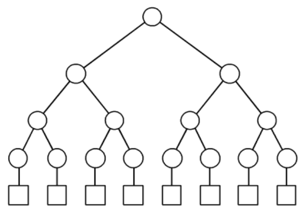
- Diameter
  - Average minumum distance between pairs of nodes
- Bisection bandwidth
  - sum of the bandwidth of the minimum number of channels which, if removed, would partition the network into two sub-graphs
- Number of edges / node
- Constant edge length? (yes/no)

2-D Meshes

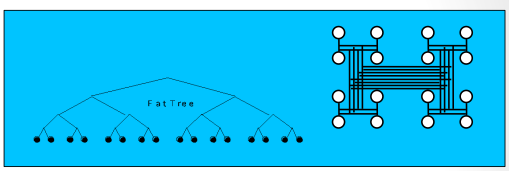


(a) (b)

Binary Tree Network



Fat-Tree



- Fatter links (really more of them) as you go up, so bisection BW scales with N
- Implemented on Janus

Thinking about distributed memory

- In terms of hardware,
  - Direct connections would require $O(p^2)$ wires and communication ports: infeasible for large p
  - Limited connectivity necessitates routing data through intermediate processors
    (consider a 2D mesh)
    - Store and forward routing: more time and storage
    - Cut through or wormhole routing
      - Both essentially transparent to users

Message passing

- Most natural and efficient paradigm for distributed-memory systems
- Two-sided, send and receive communication between processes
- Efficiently portable to shared-memory or almost any other parallel architecture:
  "assembly language of parallel computing" due to universality and detailed, low-level control of parallelism

More on message passing

- Provides natural synchronization among processes (through blocking receives, for example), so explicit synchronization of memory access is unnecessary
- Sometimes deemed tedious and low-level, but thinking about locality promotes
  - good performance,
  - scalability,
  - portability
- Dominant paradigm for developing portable and scalable applications for massively parallel systems

Programming a distributed-memory computer

- MPI (Message Passing Interface)
  also PVM (Parallel Virtual Machine) and others
- Message passing standard, universally adopted
  =
  library of communication routines
  callable from C, C++, Fortran, (Python)
- 125+ functions—we will study small subset
  may be possible to improve performance with more

MPI-1

- MPI was developed in two major stages, MPI-1 and MPI-2
- Features of MPI-1 include
  - point-to-point communication
  - collective communication process
  - groups and communication domains
  - virtual process topologies
  - environmental management and inquiry
  - profiling interface bindings for Fortran and C

MPI-2

- Additional features of MPI-2 include:
  - dynamic process management input/output
  - one-sided operations for remote memory access (update or interrogate)
  - memory access bindings for C++

- We will cover very little of MPI-2
  - not essential for algorithms we will consider
  - not supported well on some parallel systems

MPI programs use SPMD model

- Same program runs on each process
- Build executable and link with MPI library
- User determines number of processes and on which processors they will run

Programming in MPI

```
integer ierr
include "mpi.h"
call MPI_init(ierr)
.
.
.
call MPI_Finalize(ierr)
```

Yes, this is Fortran (ignoring indents)
C returns error codes as function values,
Fortran requires arguments (ierr)

Programming in MPI

```
integer ierr
include "mpi.h"
call MPI_init(ierr)
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
.
.
.
call MPI_Finalize(ierr)
```

Determine process id or *rank* (here = myid)
And number of processes (here = numprocs)

MPI_COMM_WORLD

- Is a *communicator*
- Predefined in MPI
- Consists of all processes running at start of program execution
- Process rank and number of processors determined from MPI_COMM_WORLD
- Possible to create new communicators
  - Will do this in latter in the class

Example program

- Summing numbers on ring of processors
  initially, single number per processor

- If I am processor myid,
- Store my number in x(myid+1)
- For number of steps = numprocs – 1
  - Send my result to process myid + 1 (mod numprocs)
  - Receive x(myid) from process myid – 1 (also mod numprocs)
  - Send result to myid + 1
  - and so on until values have been received from all processors
  - Once all values have been received, sum x(1)+…+x(numprocs)

Blocking send

- call MPI_SEND(

message,	e.g., my_partial_sum,
count,	number of values in msg
data_type,	e.g, MPI_DOUBLE_PRECISION,
destination,	e.g., myid + 1
tag,	some info about msg, e.g., store it
communicator,	e.g., MPI_COMM_WORLD,
ierr	

)

All arguments are inputs.

Fortran MPI Data Types

MPI_CHARACTER
MPI_COMPLEX, MPI_COMPLEX8, also 16 and 32
MPI_DOUBLE_COMPLEX
MPI_DOUBLE_PRECISION
MPI_INTEGER
MPI_INTEGER1, MPI_INTEGER2, also 4 and 8
MPI_LOGICAL
MPI_LOGICAL1, MPI_LOGICAL2, also 4 and 8
MPI_REAL
MPI_REAL4, MPI_REAL8, MPI_REAL16

Numbers = numbers of bytes
Somewhat different in C—see text or Google it

Blocking?

- Blocking send
  - does not return until the message data and envelope have been buffered in matching receive buffer or temporary system buffer.
  - can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver.
  - MPI buffers or not, depending on availability of space
  - **non-local**: successful completion of the send operation may depend on the occurrence of a matching receive.

Blocking receive

- call MPI_RECV(

message,	e.g., my_partial_sum,
count,	number of values in msg
data_type,	e.g, MPI_DOUBLE_PRECISION,
source,	e.g., myid - 1
tag,	some info about msg, e.g., store it
communicator,	e.g., MPI_COMM_WORLD,
status,	info on size of message received
ierr	

)

The arguments

- outputs: message, status
- count*size of data_type determines size of receive buffer:
  --too large message received gives error,
  --too small message is ok
- status must be decoded if needed (MPI_Get_Count)

Blocking receive

- Process must wait until message is received to return from call.
- Stalls progress of program BUT
  - blocking sends and receives enforce process synchronization
  - so enforce consistency of data

Our program

```
integer ierr  (and other dimension statements)
include "mpi.h"
call MPI_init(ierr), MPI_COMM_RANK, MPI_COMM_SIZE
< Processor myid has x(1), x(2) to begin>
count = 1
do j = 1, numprocs-1
   call MPI_send(x(count), 2, …,mod(myid+1,numprocs),…)
   count = count + 2
   call MPI_recv(x(count), 2, …, mod(myid-1,numprocs),…)
enddo
print*,'here is my answer',sum(x)
Call MPI_finalize(ierr)
```


Where to get MPI

- Custom versions of MPI supplied by vendors of almost all current parallel computers
- Freeware versions available for clusters, etc.
  - MPICH: www.mcs.anl.gov/mpi/mpich
  - OpenMPI: www.open-mpi.org
- Visit websites for tutorials on learning and using MPI
- MPI Forum (www.mpi-forum.org) has free versions of MPI standard (both MPI-1 and MPI-2), docs and archives