High Performance Parallel Programming (CS61064)

Week – 4
Part 2

Pralay Mitra

Flynn's taxonomy

Single data stream

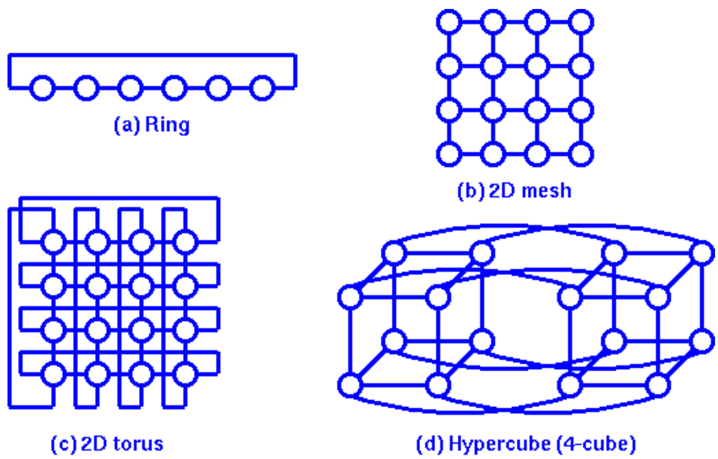
SISD MISD

Multiple data streams

SIMD MIMD SPMD MPMD

Distributed Memory

- Interconnection Network



Distributed Memory

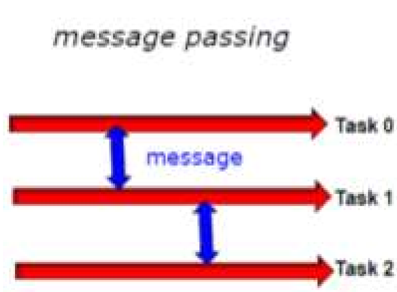
- Interconnection Network
  - Static
  - Dynamic

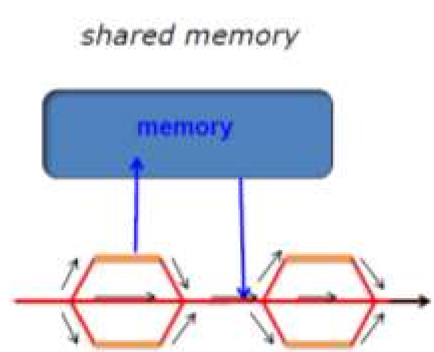
Network	Diameter	BisectionWidth	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Star	2	1	1	$p-1$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
Wraparound k-ary d-cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

MPI

Introduction

Message passing and shared memory





The Message-Passing Model

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Inter-process communication consists of
  - Synchronization
  - Movement of data from one process's address space to another's.

Types of Parallel Computing Models

- Data Parallel - the same instructions are carried out simultaneously on multiple data items (SIMD)
- Task Parallel - different instructions on different data (MIMD)
- SPMD (single program, multiple data) not synchronized at individual operation level
- SPMD is equivalent to MIMD since each MIMD program can be made SPMD (similarly for SIMD, but not in practical sense.)

Message passing (and MPI) is for MIMD/SPMD parallelism.

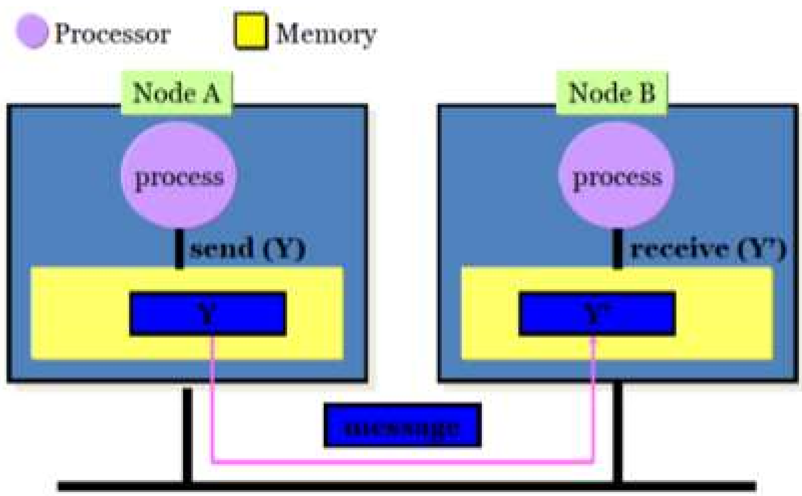
Message Passing

- Resources are local
- Each process operates in its own environment. Exchange of information occurs via communication.
- Example messages: Data, instructions, signals (synchronization)
- Message passing scheme can also be implemented in shared memory architecture.

Message Passing
Communication and Synchronization

- The sender process cooperates with the destination process.
- The communication system should allow
  - » send (message)
  - » receive (message)
  - » synchronization

MPI programming model



Why MPI?

- Portability
- Scalability
- High Performance
- Optimized for hardware
- Splitting of workload and data

Pros and Cons

- Pros
  - HPC system supports highly optimized communication hardware and software
  - Portable and scalable
  - Many applications are there
- Cons
  - Message passing
  - Most serial programs are required to rewrite
  - Memory overhead

Minimize message passing as much as possible!!!

MPI Standard

- **Prime Goals**
  - Allow efficient implementation
  - Provide source code portability
- **Additionally**
  - A great deal of functionality
  - Support for heterogeneous parallel architecture
- **Extra in MPI2**
  - Library for parallel I/O, remote memory access, multi threads, object oriented programming
- **Extra in MPI3**
  - Include non-blocking collectives, improved RMA and neighborhood collectives.

NOTE

There are various implementations (*IntelMPI*, *OpenMPI*, *MPICH*, *HPMPI* etc) which have different performance, features and standards compliance.

MPI programs

- An MPI program is the collection of multiple instances of a serial program that communicate through message passing.

- Basic features:
  - Initialize, manage and terminate communications.
  - Communicate between pairs of processors (p2p).
  - Communicate among groups of processors (collective).
  - Create data type

Your first MPI program

```
#include <stdio.h>
#include <mpi.h>

int main(int argc,char *argv[])
{
     MPI_Init(&argc,&argv);
     printf("Hello World!\n");
     MPI_Finalize();
     return 0;
}
```

Compilation and Execution

- Compilation
  $ mpicc hello.c

- Execution
  - Interactive

```
$ mpirun -np 8 ./a.out        // mpirun --mca btl ^uct -np 6 ./a.out
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Job Scheduler and Workload Manager

- Interactive job
- Batch System

- Submit a job
  - $ qsub <scriptname>

- Check the status of a job
  - $ qstat

- Delete a job
  - $ qdel <jobid>

Sample PBS script

```
# declare a name for this job to be sample_job
#PBS -N myFirst_job
# request the queue
#PBS -q  high
# request node
#PBS -l nodes=64
# request 128 hours wall clock time
#PBS -l walltime=128:00:00
# mail is sent to you when the job starts and when it terminates or aborts
#PBS -m bea
# specify your email address
#PBS -M pralay@cse.iitkgp.ac.in
# By default, PBS scripts execute in your home directory,
cd $PBS_O_WORKDIR
# run the program
mpirun ./a.out
exit 0
```

Status of the queue

```
Jobid     Name     User       Time     S Queue
----------------------------------------------
7520.pro eodesign pralay     15:45:05 R high
7452.pro ppdock   anupam     12:40:56 R high
8246.pro MCsimula barnali    0        Q low
```

Compilation and Execution

- Compilation
  $ mpicc hello.c

- Execution
  - **Batch**
    $ bsub script.sh
    Job <9068> is submitted to default queue <low>.
    $ ls *9068*
    err.9068 out.9068

MPI Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **`MPI_Comm_size`** reports the number of processes.
  - **`MPI_Comm_rank`** reports the *rank*, a number between 0 and size-1, identifying the calling process

Your second MPI program

```
#include <stdio.h>
#include <mpi.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```