SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

PARALLEL & DISTRIBUTED SYSTEMS LABORATORY

Course Code: CSE411

Semester: VII

Lab Manual

2025

SHANMUGHA ARTS, SCIENCE, TECHNOLOGY AND RESEARCH ACADEMY
(SASTRA Deemed to be) University
Tirumalaisamudram, Thanjavur-613 401
School of Computing

Course Objective: The course will help the learner to implement parallel algorithms using CUDA and develop applications for distributed computing concepts using MPI.

Course Learning Outcomes:

Upon successful completion of this course, the learner will be able to:

- Design basic parallel programs in CUDA using parallel reduction
- Use various CUDA memories and implement PRAM algorithms such as matrix multiplication and sorting
- Create chat server application for multiple clients in a distributed environment
- Employ MPI programming to implement group communication and clock synchronization concepts with multiple processors
- Develop leader election algorithm with multiple processors
- Experiment with fault tolerance mechanism using byzantine agreement

List of Experiments:

- Compute dot-product of two vectors using parallel reduction.
- Program on matrix transpose using shared memory.
- Compute 1-D Stencil using constant memory.
- Program to implement pre-order tree traversal using PRAM algorithm.
- Program to implement odd-even transposition sort
- Program to implement quick sort algorithm.
- Design chat server application with multiple clients.
- Program to implement mutual exclusion in distributed environment.
- Program to implement group communication.
- Program to implement clock synchronization.
- Program to demonstrate leader election algorithm.
- Program to implement fault tolerance mechanism using Byzantine agreement.

Exercise No. 1 Dot Product of Two Vectors

Write a program to compute dot product of two vectors using CUDA and OpenMP.

Objective

Develop a high-performance program to compute the dot product of two vectors using CUDA for GPU parallelization and OpenMP for CPU parallelization. The goal is to compare the performance of both parallel computing techniques in terms of execution speed and efficiency.

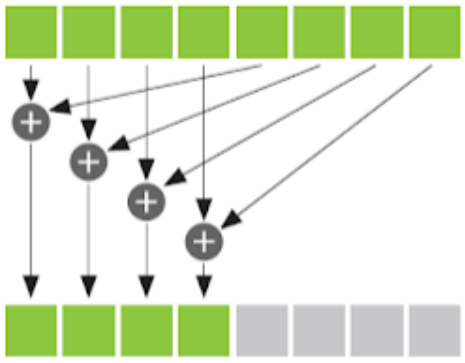
Prerequisite : Solid understanding of arrays and pointers.

Concept

- **Dot Product:** The dot product (or scalar product) of two vectors is a fundamental operation in linear algebra. For two vectors **a** and **b** of length n:
  dot product = $\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i \cdot b_i$, Where a_i and b_i are the elements of the vectors **a** and **b**, respectively.

  $$c = a \cdot b$$
  $$c = (a0, a1, a2, a3) \cdot (b0, b1, b2, b3)$$
  $$c = a0\ b0 + a1\ b1 + a2\ b2 + a3\ b3$$

- **Parallel Reduction:** The original problem is divided into smaller sub-problems that can be solved concurrently. The intermediate results of these sub-problems are then combined to form the final result. This process involves multiple stages or steps, each reducing the number of elements by half, typically using a binary tree structure.



- **CUDA Implementation:** In CUDA, the dot product is computed by dividing the work among many threads running on the GPU. Each thread computes a partial sum of the dot product. These partial sums are then combined using atomic operations to ensure concurrent updates are performed without conflicts.

- **OpenMP Implementation:** In OpenMP, the dot product is computed by dividing the work among multiple threads running on the CPU. Each thread computes a portion of the dot product and then combines the results using a reduction operation.

Procedure

- Input two vectors a and b of length n.
- Allocate memory on the GPU for vectors a, b, and an output array. Copy a and b from host to device.
- Define a CUDA kernel to compute the dot product using parallel reduction and Use atomic operations to combine results across blocks.
- Retrieve the final result from the device and free allocated memory.
- Measure the execution time using CudaEvent API.
- Follow similar steps for OpenMP program and get the execution time.

Sample Input

n = 1<< 20, a[ ]={1,1,1,….,1} , b[ ]={2,2,2,…,2}

Output

Dot product : 2000000

Execution time (CUDA): X.XXXXXXs

Execution time (OpenMP): X.XXXXXXs

Exercise No. 2 Matrix Transpose using Shared Memory

Write CUDA and OpenMP program to compute the transpose of a given matrix using shared memory.

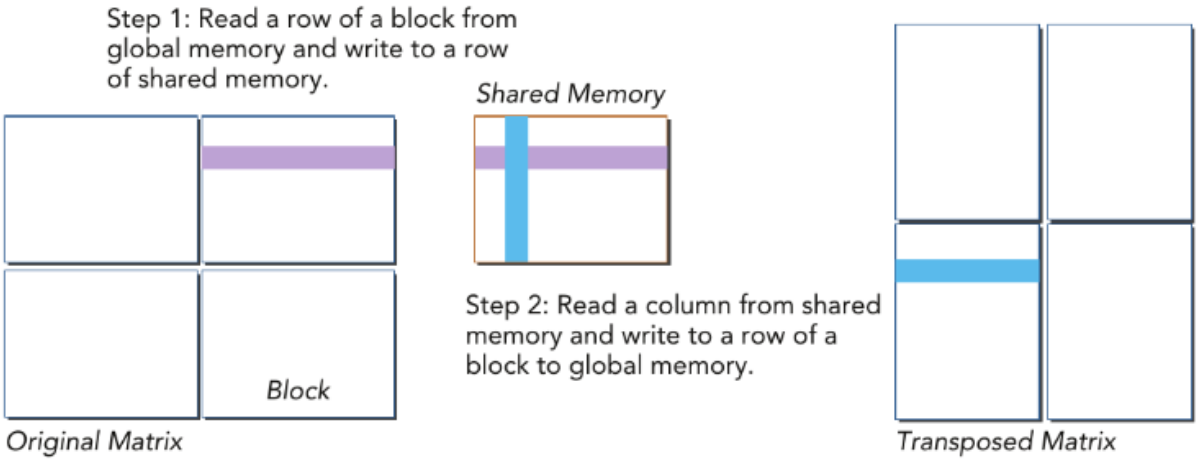
Objective

Implement CUDA and OpenMP programs to compute the transpose of a given matrix, leveraging shared memory to minimize latency and enhance computational efficiency. Measure and compare the execution times of CUDA and OpenMP implementations to assess the performance improvements.

Prerequisite: Familiarity with matrix data structures, indexing, and basic operations (e.g., addition, multiplication). Also, require knowledge of parallelism, threads, and synchronization mechanisms.

Concept

Shared memory is limited and shared among threads in the same block, making it faster than global memory. Using shared memory in CUDA can significantly improve performance by reducing the number of global memory accesses.



Procedure

- Define the matrix dimensions and allocate memory for the input matrix and the transposed matrix on both the host (CPU) and device (GPU).
- Define a CUDA kernel that allocates shared memory for a tile of the matrix, loads the matrix elements into the shared memory tile, synchronizes threads to ensure all elements are loaded, and writes the transposed elements from shared memory back to the global memory.

- Configure the grid and block dimensions, launch the CUDA kernel to perform the transposition and copy the transposed matrix from device memory back to host memory.
- Free the allocated memory on the host and device.
- Follow similar steps for OpenMP program.

Sample Input

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Output

$\mathbf{Transposed\ matrix}$: $\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$

Execution time (CUDA): X.XXXXXXs

Execution time (OpenMP): X.XXXXXXs

Exercise No. 3 1-D Stencil using Constant Memory

Write a CUDA program to compute the 1-D Stencil using Constant Memory.

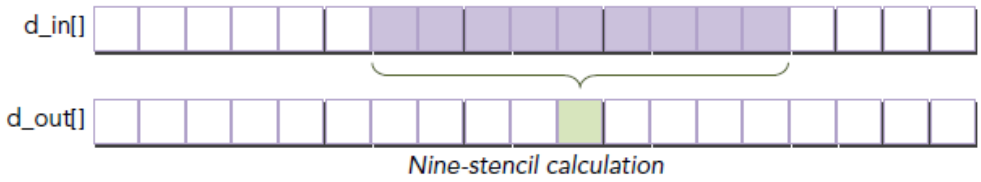
Objective

Implement a CUDA program to compute the 1-D stencil operation, leveraging constant memory to reduce global memory access latency and improve computational efficiency. Measure the execution time and compare the benefits of using constant memory to standard global memory access.

Prerequisite: Familiarity with constant memory benefits and access patterns, and a solid understanding of the 1-D stencil operation and its computational requirements.

Concept

- In the numerical analysis domain, a stencil computation applies a function to a collection of geometric points and updates the value of a single point with the output.
- Stencils are the basis for many algorithms that solve partial differential equations. In 1D, a nine-point stencil around a point at position x would apply some function to the values at these positions: $\{x - 4h, x - 3h, x - 2h, x - h, x, x + h, x + 2h, x + 3h, x + 4h\}$
- An example of a nine-point stencil is the eighth-order centered difference formula for the first derivative of a function f of a real variable at a point x.
- $f'(x) \approx c0\,(f(x + 4h) - f(x - 4h)) + c1\,(f(x + 3h) - f(x - 3h)) - c2\,(f(x + 2h) - f(x - 2h)) + c3\,(f(x + h) - f(x - h))$



Nine-stencil calculation

- Applying this formula across a 1D array is a data parallel operation that maps well to CUDA. Assign a position x to each thread and have it calculate $f'(x)$.

Procedure

- Allocate host and device memory for the input array and the output array. Initialize the input array with data and copy it to the device.
- Define the stencil coefficients in the host code and Copy the stencil coefficients to CUDA constant memory.
- Implement a CUDA kernel that performs the stencil computation.

- Each thread will read the relevant elements from the input array and apply the stencil operation using the coefficients stored in constant memory.
- Configure the kernel launch parameters (number of blocks and threads per block) and Launch the kernel to perform the stencil computation.
- Copy the output array from the device to the host and Free the allocated device memory.
- Use CUDA events to measure the execution time of the kernel. Compare the performance of using constant memory with standard global memory access.

Sample Input

Input Array: [1, 2, 3, 4**, 5, 6, 7, 8, 9, 10, 11, 12**, 13, 14, 15, 16]

Coefficients: [c0, c1, c2, c3]

Output

Execution time (Global memory): X.XXXXXXs

Execution time (Constant memory): X.XXXXXXs

Exercise No. 4 Preorder tree traversal

Write a CUDA program to implement preorder tree traversal CREW PRAM algorithm.

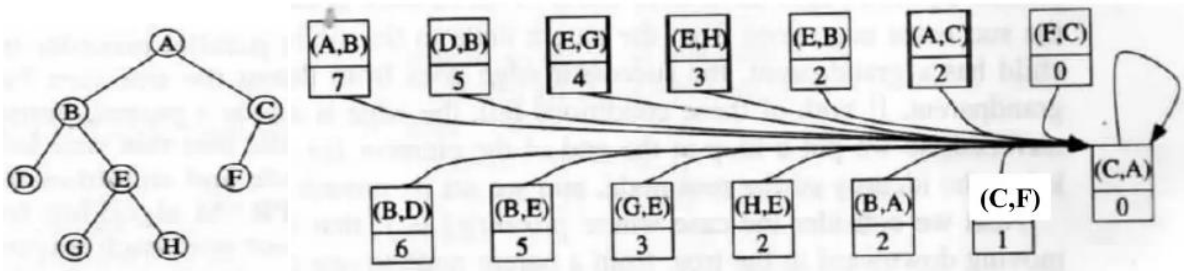
Objective

Develop a CUDA program that efficiently performs preorder tree traversal by leveraging parallel processing capabilities and following the list ranking technique under the CREW PRAM model. The implementation will demonstrate the use of parallel algorithms for tree data structures and the benefits of using CUDA for such computations.

Prerequisite: Understanding of binary trees and preorder tree traversal algorithm. Familiarity with parallel processing concepts and algorithms, particularly the list ranking technique and suffix sum calculations.

Concept

- Preorder traversal visits nodes in the following order: root, left subtree, right subtree.
- List ranking is a fundamental parallel algorithm that assigns a rank or position to each element in a linked list. In the context of tree traversal, list ranking helps in determining the order of nodes to be visited during the traversal.
- Suffix sum is a parallel algorithm used to compute the sum of all elements that appear after a given element in a list. For preorder traversal, suffix sum helps in calculating the rank of each node in the traversal order.
- The CREW PRAM model allows concurrent reading of the same memory location by multiple processors, but only one processor can write to a memory location at a time.



Vertices	A	B	C	D	E	F	G	H
Positions	1	7	2	6	5	1	4	3
Prorder	1	2	7	3	4	8	5	6

Procedure

- Represent the binary tree as an array.
- Construct singly linked list for the given tree based on edges.
    - If parent[i]=j, the edge is moving upward in the tree, from a child node to its parent.
    - If the child has a sibling, the successor edge goes from the parent node to the sibling.
    - Otherwise, if the child has a grandparent, the successor edge goes from the parent node to the grand parent.
    - If both of these conditions fail, the edge is at the end of the tree traversal and Preorder number is initialized to 1.
- Assign position to each node

  if parent[i]=j then position(i,j) = 0

  else position(i,j) = 1
- Compute Suffix Sum

  for k=1 to log(2(n-1)) do

  postion(i,j)=postion(i,j) + postion[succ[(i,j)]] succ[(i,j)] =succ[succ[(i,j)]]
- Compute Preorder for each node

  if i=parent[j] then prorder[j]=n+1- postion(i,j)
- Print the nodes of the tree in preorder traversal order.

Sample Input

Binary tree with nodes: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

Output

Preorder Traversal: 1 2 4 8 9 5 10 11 3 6 12 13 7 14 15

Exercise No. 5 Odd-Even Transposition Sort

Write a CUDA program to sort the numbers using Odd-Even Transposition Sort

Objective

Implement a CUDA program to perform sorting using the odd-even transposition sort algorithm, leveraging GPU parallelism for improved performance. Measure execution time and compare with CPU-based sorting algorithms.

Prerequisite: Familiarity with the odd-even transposition sort and its application in parallel computing.

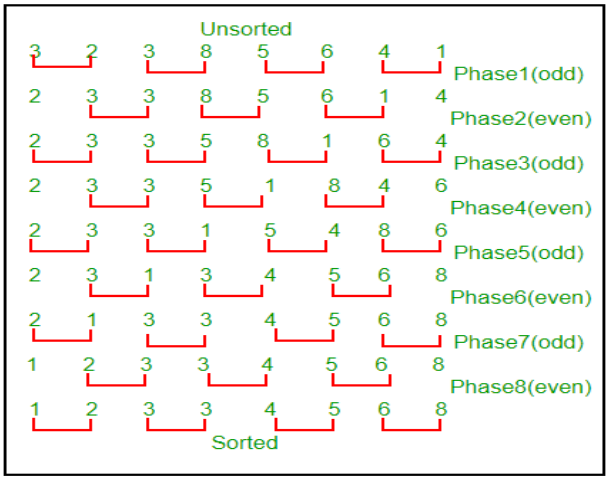
Concept

The odd-even transposition sort is a parallel sorting algorithm suitable for distributed memory architectures. In CUDA, it can be efficiently implemented using thread parallelism within a GPU block, where each thread compares and swaps elements based on its index and the current phase (odd or even).

Procedure

- Allocate memory for the input and output arrays on the GPU. Read the input array of numbers to be sorted.
- Implement a CUDA kernel (odd_even_sort) that performs the odd-even transposition sort. Utilize thread indices to handle comparisons and swaps between adjacent elements.

```
if j<n-1 and odd(j) then
        t=successor(a)
        successor(a)=max(a,t)
        a=min(a,t)
endif
if even(j) then
        t=successor(a)
        successor(a)=max(a,t)
        a=min(a,t)
endif
```



- Use __syncthreads() to synchronize threads within a block after each phase to maintain sorting order.
- Iterate kernel launches for alternating odd and even phases until convergence (sorted order).
- Copy the sorted array from GPU device memory back to the host.
- Validate the sorted GPU output array by comparing with output of CPU based sorting algorithm.
- Use CUDA events to measure the execution time and compare with CPU sorting.

Sample Input

N = 1<< 20, Initialize the array with random numbers.

Output

Execution time (GPU): X.XXXXXXs

Execution time (CPU): X.XXXXXXs

Exercise No. 6 Parallel Quick Sort

Write a CUDA program to implement parallel Quick sort algorithm

Objective

Implement a CUDA program for parallel quicksort, leveraging global stack management to dynamically distribute unsorted subarrays among threads.

Prerequisite: Familiarity with quicksort algorithm and insertion sort and also require Knowledge of stack data structure and mutual exclusion mechanisms in CUDA.

Concept

The program maintains a global stack protected by atomic operations for mutual exclusivity. Each thread retrieves bounds from the stack and performs partitioning, and applies insertion sort for small subarrays. The threads continue until all elements are sorted.

Procedure

- Initialize the input array in global memory. Allocate memory for the global stack to store subarray bounds.
- Implement Kernel for Parallel Quicksort:

```
sorted=0
INITIALIZE.STACK()
for all Pi, where 0 <= i <p do
  while (sorted<n) do
    bounds = STACK.DELETE()
    while (bounds.low < bounds.high) do
      median = PARTITION (bounds.low, bounds.high)
      STACK.INSERT (median + 1, bounds.high)
      bounds.high = median - 1
      if bounds.low = bounds.high then
        ADD.TO.SORTED (2)
      else
        ADD.TO.SORTED (1)
       endif
     endwhile
  endwhile
endfor
```

- Launch the kernel with an appropriate number of threads.

- Ensure that idle threads fetch work from the stack until all elements are sorted.
- Copy the sorted array back to the host.
- Validate the sorted GPU output array by comparing with output of CPU based sorting algorithm
- Use CUDA events to measure the execution time and compare with CPU sorting.

Sample Input

N = 1<< 10, Iinitialize the array with random numbers.

Output

Execution time (GPU): X.XXXXXXs

Execution time (CPU): X.XXXXXXs

Exercise No. 7 Chat server application with multiple clients

Write an MPI program to implement a Chat server with multiple clients.

Objective

Write an MPI program to implement a chat server where a root process broadcasts a message to all other processes, and processes can communicate with each other using MPI_Send and MPI_Recv.

Prerequisite: Familiarity with the Message Passing Interface (MPI) and its fundamental functions such as MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Send, MPI_Recv, and MPI_Finalize.

Concept

MPI is a standardized and portable message-passing system designed to function on parallel computing architectures. It allows processes to communicate with each other by sending and receiving messages.

In a typical MPI program:

- Initialization and Finalization: All MPI programs start with MPI_Init and end with MPI_Finalize.
- Rank and Size: Each process in an MPI program is assigned a unique identifier called rank, and the total number of processes is referred to as the size of the communicator.
- Point-to-Point Communication: This includes sending and receiving messages between pairs of processes using MPI_Send and MPI_Recv.
- Broadcasting: A message from one process (usually the root) is sent to all other processes in the communicator using MPI_Bcast.

Procedure

- Create multiple processors.
- Broadcast a message from the process with rank "root" to all other processes in the group.
- Every processor in the group receives the message from the root using MPI_Recv.
- Communication between processors is done using MPI_Send.
- Demonstrate the implementation with at least 4 processors.

Sample Input

mpicc −o mpi chatserver.c

mpirun −np 4 mpi

Output

Process 0 broadcasting message: Hello from root!

Process 1 received broadcast message: Hello from root!

Process 2 received broadcast message: Hello from root!

Process 3 received broadcast message: Hello from root!

Root process received message: Hello from process 1

Root process received message: Hello from process 2

Root process received message: Hello from process 3

Process 2 received message: Message from process 1 to process 2

Exercise No. 8 Mutual Exclusion

Write an MPI program to implement Mutual exclusion.

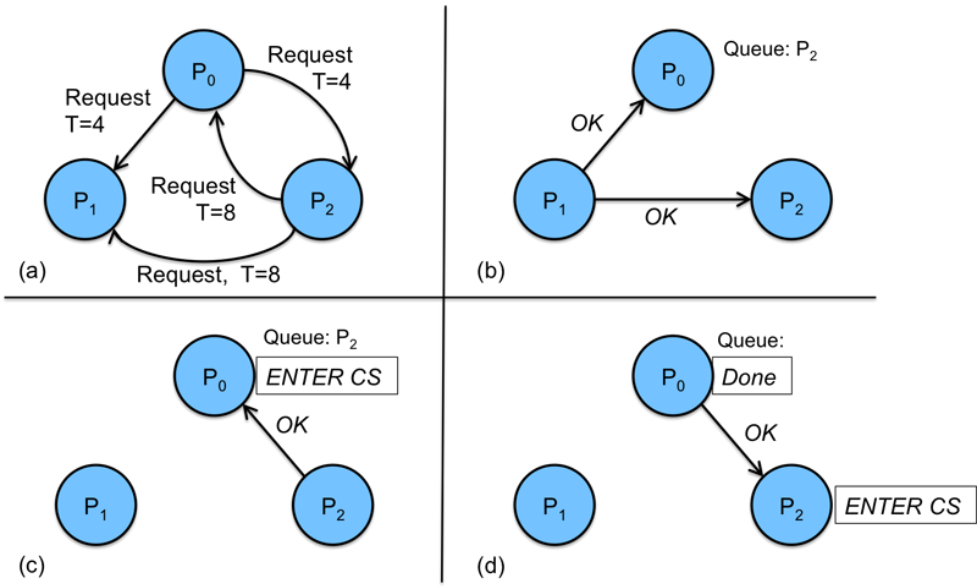
Objective

Implement an MPI program to achieve mutual exclusion in a distributed system using the Ricart-Agrawala algorithm.

Prerequisite: Familiarity with the MPI functions such as MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Send, MPI_Recv, and MPI_Finalize, along with an understanding of mutual exclusion and distributed systems.

Concept

- Mutual Exclusion: Ensures that only one process enters the critical section at a time.
- Ricart-Agrawala Algorithm: A decentralized algorithm that uses a request-reply mechanism to achieve mutual exclusion.
    - When a process wants to enter the critical section, it sends a request message to all other processes.
    - Each process replies to the request if it is not in the critical section or if it has a lower priority request.
    - The requesting process enters the critical section only after receiving replies from all other processes.



Procedure

- Initialize MPI environment.
- Each process sends a request message to all other processes when it wants to enter the critical section.
- Each process replies to the request if it is not in the critical section or has a lower priority request.
- The requesting process enters the critical section after receiving replies from all other processes.
- When the process exits the critical section, it sends a release message to all other processes.

Sample Input

mpicc −o mpi mutualexclusion.c

mpirun −np 4 mpi

Output

Process 0 requesting critical section

Process 0 received REQUEST from process 1

Process 0 received REQUEST from process 2

Process 0 received REQUEST from process 3

Process 0 received OK from process 1

Process 0 received OK from process 2

Process 0 received OK from process 3

Process 0 in critical section

Process 0 releasing critical section

Process 1 received RELEASE from process 0

Process 2 received RELEASE from process 0

Process 3 received RELEASE from process 0

Exercise No. 9 Group Communication

Write an MPI program to implement group communication.

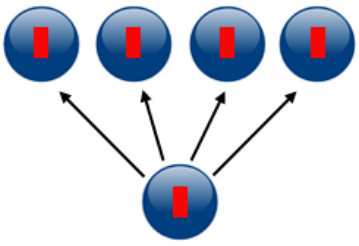
Objective

Implement an MPI program to demonstrate group communication using collective communication functions.

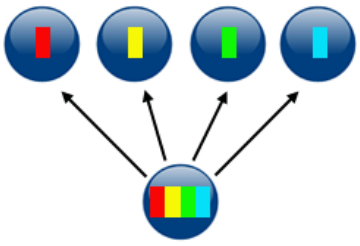
Prerequisite: Familiarity with MPI functions such as MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Reduce, and MPI_Finalize.

Concept

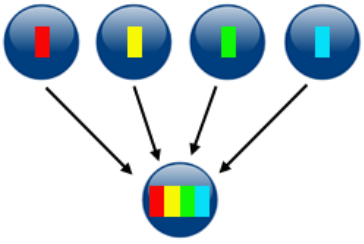
- Broadcast (MPI_Bcast): Distributes a message from a single source to all tasks in a group.
- Scatter (MPI_Scatter): Distributes distinct messages from a single source to all tasks in a group.
- Gather (MPI_Gather): Gathers distinct messages from each task in the group into a single destination task.
- Reduce (MPI_Reduce): Combines values from all processes and returns the result to the root process.



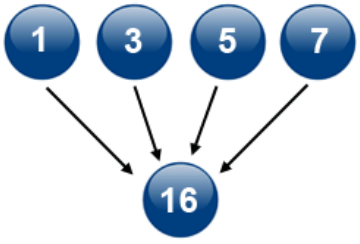
broadcast



scatter



gather



reduction

Procedure

- Initialize MPI environment.
- Determine the rank and size of processes.
- Use MPI_Bcast to distribute a message from the root process to all other processes.
- Use MPI_Scatter to distribute distinct messages from the root process to all other processes.
- Use MPI_Gather to gather messages from all processes into the root process.
- Use MPI_Reduce to perform a reduction operation (e.g., summation) on values from all processes and send the result to the root process.
- Finalize the MPI environment.

Sample Input

mpicc −o mpi groupcommunication.c

mpirun −np 4 mpi

Output

Process 0 broadcasts 100

Process 1 received 100 from bcast

Process 2 received 100 from bcast

Process 3 received 100 from bcast

Process 0 received 10 from scatter

Process 1 received 20 from scatter

Process 2 received 30 from scatter

Process 3 received 40 from scatter

Root process gathered data: 10 20 30 40

Reduction result: 100

Exercise No. 10 Clock Synchronization

Write an MPI program to implement Clock Synchronization algorithm.

Objective

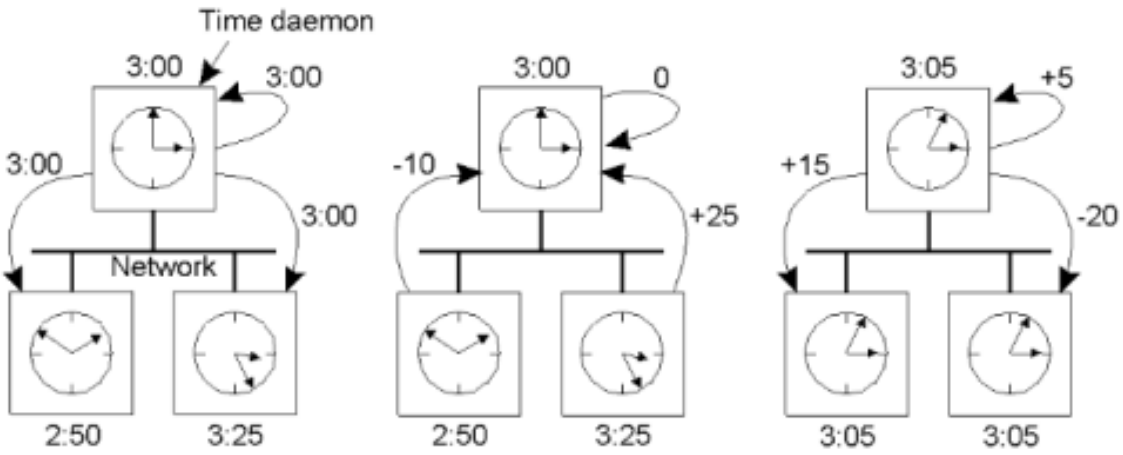
Implement an MPI program to demonstrate the Clock Synchronization using the Berkeley algorithm.

Prerequisite: Familiarity with MPI functions such as MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Send, MPI_Recv, and MPI_Finalize. Also, an understanding of Berkeley Clock synchronization algorithm.

Concept

Clock Synchronization (Berkeley Algorithm): The Berkeley algorithm is a method to synchronize the clocks of multiple processors in a distributed system. It works as follows:

- A designated process, called the master, polls the time from all other processes.
- The master calculates the average time (considering all reported times).
- The master sends each process the amount they need to adjust their clocks by, to align with the average time.



Procedure

- **I**nitialize MPI environment.
- Determine the rank and size of processes.
- The master process collects time from all other processes.
- The master computes the average time and the necessary adjustment for each process.
- The master sends the time adjustment to all processes.
- Each process adjusts its clock based on the received adjustment.
- Finalize the MPI environment.

Sample Input

mpicc −o mpi clocksync.c

mpirun −np 4 mpi

Output

Process 0 local time: 1628775600

Process 1 local time: 1628775610

Process 2 local time: 1628775620

Process 3 local time: 1628775630

Average time: 1628775615

Process 0 adjusted time: 1628775615

Process 1 adjusted time: 1628775615

Process 2 adjusted time: 1628775615

Process 3 adjusted time: 1628775615

Exercise No. 11 Leader Election

Write an MPI program to implement Leader Election algorithm for a Ring network.

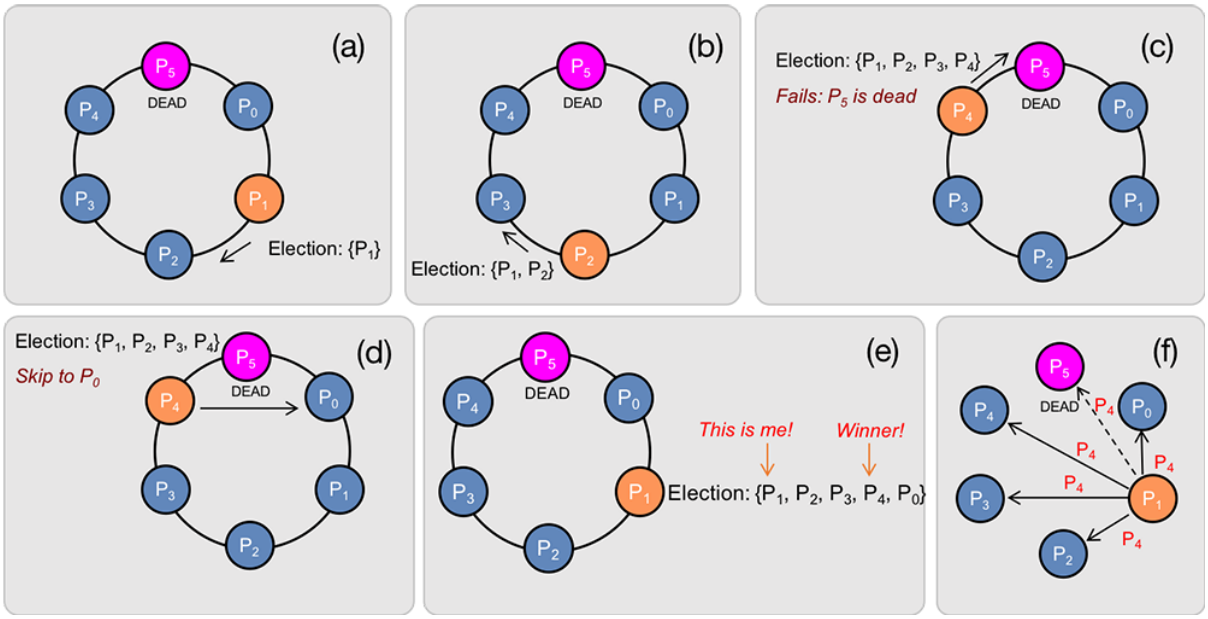
Objective

Implement an MPI program to Implement an MPI program to demonstrate the Leader Election algorithm for a Ring network where the process that detects the failure of the current leader initiates a new election.

Prerequisite: Familiarity with MPI functions such as MPI_Init, MPI_Comm_rank, MPI_Comm_size, MPI_Send, MPI_Recv, and MPI_Finalize, along with the understanding of leader election algorithm for a ring network.

Concept

In this algorithm, a process that detects the failure of the leader initiates an election. It sends an election message containing its rank to the next process in the ring. Each process forwards the received election message and keeps track of the highest rank seen. When the election message returns to the initiator, it announces the process with the highest rank as the new leader



Procedure

- Initialize the MPI environment.
- Determine the rank and size of processes.
- Each process sends its rank in an election message when initiating or forwarding an election.

- Each process forwards the election message around the ring, keeping track of the highest rank.
- When the election message returns to the initiator, it announces the process with the highest rank as the new leader.
- The new leader is broadcast to all processes.
- Finalize the MPI environment.

Sample Input

mpicc −o mpi clocksync.c

mpirun −np 6 mpi

Output

Process 1 Detected failure of coordinator process 5 and initiated an election.

Process 1: Coordinator is process 4

Process 2: Coordinator is process 4

Process 3: Coordinator is process 4

Process 4: Coordinator is process 4

Process 0: Coordinator is process 4

Exercise No. 12 Byzantine Agreement Problem

Write an MPI program to implement Byzantine Agreement problem

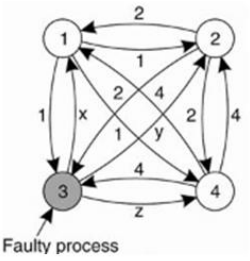
Objective

Write an MPI program to implement the Byzantine Agreement problem, where non-faulty processes reach consensus on a value in the presence of a faulty process.

Prerequisite: Familiarity with MPI functions and understanding of the Byzantine Agreement problem and distributed consensus.

Concept

The problem involves reaching consensus among distributed processes (nodes) in the presence of faulty or malicious nodes. The goal is for all non-faulty nodes to agree on the same value, even if some nodes are faulty and may send conflicting or incorrect information.



1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

1 Got	2 Got	4 Got
(1,2,y,4)	(1,2,x,4)	(1,2,x,4)
(1,2,3,4)	(1,2,3,4)	(1,2,y,4)
(1,2,z,4)	(1,2,z,4)	(1,2,3,4)

Procedure

- Initialize the MPI environment.
- Determine the rank and size of processes.
- Each process generates a random value and sends its value to all other processes.
- Each process collects values from all other processes.
- Each process performs a majority vote to agree on a single value.
- Finalize the MPI environment.

Sample Input

mpicc −o mpi faulty.c

mpirun −np 6 mpi

Output

Process 0: 3 is the faulty process

Process 1: 3 is the faulty process

Process 2: 3 is the faulty process

Process 4: 3 is the faulty process

Additional Exercises:

1. Write a program to perform Enumeration Sort
2. Write a parallel program to perform (i) Suffix sum (ii) Prefix sum
3. Write a parallel program to Generate Prime Numbers from 1 to 1000
4. Write a parallel program to perform Bitonic Merge Sort
5. Write a program to Detect Distributed Deadlocks
6. Write a program to implement Dijkstra's Shortest Path Algorithm
7. Write a parallel program to perform block matrix multiplication using OpenMP