Message Passing Interface

Tran Hung Thinh - BI12-428

December 20, 2024

1 MPI Implementation Choice

The code utilizes the MPI (Message Passing Interface) library for implementing parallel computing across multiple processes. The MPI library provides a standardized and efficient way for processes to communicate and synchronize with each other in a distributed computing environment. The choice of MPI is suitable for parallelizing tasks across multiple processes, which is essential for the distributed system described in the code.

2 Design of MPI Service

Since the provided code implements a distributed system for leader election and location selection, the design of the MPI service revolves around message passing and synchronization between processes. The MPI service includes functions for sending and receiving messages related to participation, leader voting, and location voting. These functions facilitate communication between processes to coordinate the leader election and location selection processes.

3 Organization of the System

The system consists of multiple MPI processes running concurrently, each representing a node in the distributed system. Processes communicate with each other using MPI communication primitives such as MPI_Send and MPI_Recv. The system organization includes processes interacting with each other through message passing to achieve the desired distributed computing tasks.

4 File Transfer Implementation

In the context of the provided code, file transfer may not be directly implemented. However, if file transfer were part of the system requirements, it could be implemented using MPI I/O functions for parallel file I/O operations. Processes could collectively read from or write to files in parallel using MPI_File_read and MPI_File_write functions.

5 Task Assignment

Each MPI process performs specific tasks within the distributed system:

- Sending participation info, leader votes, and location votes.
- Receiving participation info, leader votes, and location votes.
- Choosing leaders based on received votes.
- Choosing a location based on received votes.

Each process collaborates with others to complete the overall tasks of leader election and location selection through message passing and synchronization.

6 MPI Code

```
#include <mpi.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctime>
#include <random>
#include <unistd.h>
#define ROOT 0
#define PARTICIPATION_TAG 100
#define LEADER_TAG 200
#define LOCATION_TAG 300

int updated_timer(int current, int received) {
  if (received >= current) return received + 1;
  else return current + 1;
}

void send_message(int value, int timer, int recipient, int tag) {
  int message[2] = {value, timer};
  MPI_Send(message, 2, MPI_INT, recipient, tag, MPI_COMM_WORLD);
}

void send_participation_info(int rank, int size, int &timer) {
  int participates = rand() % 2 == 0;
  for (int i = 0; i < size; i++) {
    send_message(participates, timer, i, PARTICIPATION_TAG);

    timer++;

    printf("Time = %d, Id = %d, sent participation info: %d\n",
      timer, rank, participates);
  }
}

void send_leader_vote(int rank, int size, int &timer) {
  int leader_vote = rand() % size;

  for (int recipient = 0; recipient < size; recipient++) {
```

```
    send_message(leader_vote, timer, recipient, LEADER_TAG);

    timer++;

    printf("Time = %d, Id = %d, sent leader vote: %d\n",
      timer, rank, leader_vote);
  }
}

int *receive_leaders_votes(int rank, int size, int &timer) {
  MPI_Status status;

  int *leaders_votes = new int[size];
  int message[2];

  for (int sender = 0; sender < size; sender++) {
    MPI_Recv(
      message, 2, MPI_INT,
      sender, LEADER_TAG, MPI_COMM_WORLD, &status
    );

    leaders_votes[sender] = message[0];
    timer = updated_timer(timer, message[1]);

    printf("Time = %d, Id = %d, received leader vote: %d\n",
      timer, rank, leaders_votes[sender]);
  }

  return leaders_votes;
}


void choose_leaders(int rank, int size, int &timer) {
  int *leaders_votes = receive_leaders_votes(rank, size, timer);

  int leaders[3],
      max[2] = { 0 },
      i;
  int *count_votes = new int[size]();

  for (i = 0; i < size; i++) {
    count_votes[leaders_votes[i]]++;
  }

  for (i = 0; i < 3; i++) {
    for (int j = 0; j < size; j++) {
      if (count_votes[j] > max[1]) {
        max[0] = j;
        max[1] = count_votes[j];
      }
    }
    leaders[i] = max[0];
    count_votes[max[0]] = 0;
    max[0] = 0;
    max[1] = 0;
  }

```

```
  printf("Time = %d, Id = %d, chosen leaders: [%d, %d, %d]\n",
    timer, rank, leaders[0], leaders[1], leaders[2]);

  delete[] count_votes;
  delete[] leaders_votes;
}

void send_location_vote(int size, int rank, int &timer) {
  int location_vote = rand() % 4;

  for (int recipient = 0; recipient < size; recipient++) {
    send_message(location_vote, timer, recipient, LOCATION_TAG);

    timer++;

    printf("Time = %d, Id = %d, sent location vote: %d\n",
      timer, rank, location_vote);
  }
}

int *receive_location_votes(int size, int rank, int &timer) {
  MPI_Status status;
  int *location_votes = new int[size];
  int message[2];

  for (int sender = 0; sender < size; sender++) {
    MPI_Recv(
      message, 2, MPI_INT,
      sender, LOCATION_TAG, MPI_COMM_WORLD, &status
    );
    location_votes[sender] = message[0];

    timer = updated_timer(timer, message[1]);

    printf("Time = %d, Id = %d, received location vote: %d\n",
      timer, rank, message[0]);
  }

  return location_votes;
}

void choose_location(int size, int rank, int &timer) {
  int *location_votes = receive_location_votes(size, rank, timer);

  int location = 0, max = 0, count[4] = {0}, i;

  for (i = 0; i < size; i++) {
    count[location_votes[i]]++;
  }

  for (i = 0; i < 4; i++) {
    if (count[i] > max) {
      location = i;
      max = count[i];
    }
  }

```

```
  printf("Time = %d, Id = %d, chosen location: %d\n", timer, rank,
    location);

  delete[] location_votes;
}


int main(int argc, char **argv)
{
  int size,rank;
  int timer = 0;
  int round = 1;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  srand(rank * time(NULL));

  if (size < 3) {
    printf("Not enough processes.");
    exit(0);
  }

  while(1) {
    printf("Round %d\n", round);

    send_participation_info(rank, size, timer);

    send_leader_vote(rank, size, timer);
    choose_leaders(rank, size, timer);

    send_location_vote(size, rank, timer);
    choose_location(size, rank, timer);

    printf("End of round %d\n\n\n", round);
    round++;

    usleep(5000000);
  }

  MPI_Finalize();
}
```

Listing 1: Message Passing Interface C++ Code