МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В.Ломоносова

Факультет вычислительной математики и кибернетики

Компьютерный практикум по учебному курсу

«Распределенные системы»

Разработка программы, использующей децентрализованный алгоритм с временными метками для прохождения всеми процессами критических секций

ОТЧЕТ

о выполненном задании

студента 428 учебной группы факультета ВМК МГУ

Тулина Дмитрия Ильича

гор. Москва
2022 год

Содержание

Постановка задачи **2**

Алгоритм Ricart-Agrawala **2**

Временная оценка **3**

Компиляция и запуск программы **3**

Описание программы **3**

Код программы **5**

Выводы **8**

Постановка задачи

Все 64 процессов, находящихся на разных ЭВМ сети с шинной организацией, одновременно выдали запрос на вход в критическую секцию. Реализовать программу, использующую децентрализованный алгоритм с временными метками для прохождения всеми процессами критических секций.
Критическая секция:
<проверка наличия файла “critical.txt”>;
if (<файл “critical.txt” существует>)
<сообщение об ошибке>;
<завершение работы программы>;
else
<создание файла “critical.txt”>;
Sleep (<случайное время>);
<уничтожение файла “critical.txt”>;

Для межпроцессорных взаимодействий использовать средства MPI. Получить временную оценку работы алгоритма. Оценить сколько времени потребуется для прохождения всеми критических секций. Время старта (время «разгона» после получения доступа к шине для передачи сообщения) равно 100, время передачи байта равно 1 (Ts=100,Tb=1). Доступ к шине ЭВМ получают последовательно в порядке выдачи запроса на передачу (при одновременных запросах - в порядке номеров ЭВМ). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

Алгоритм Ricart-Agrawala

Вход в критическую секцию

Когда процесс желает войти в критическую секцию, он посылает всем процессам сообщение-запрос, содержащее имя критической секции, номер процесса и текущее время.

После посылки запроса процесс ждет, пока все дадут ему разрешение. После получения от всех разрешения, он входит в критическую секцию.

Поведение процесса при приеме запроса Когда процесс получает сообщение-запрос, в зависимости от своего состояния по отношению к указанной критической секции он действует одним из следующих способов.

1. Если получатель не находится внутри критической секции и не запрашивал разрешение на вход в нее, то он посылает отправителю сообщение «OK».

2. Если получатель находится внутри критической секции, то он не отвечает, а запоминает запрос.

3. Если получатель выдал запрос на вхождение в эту секцию, но еще не вошел в нее, то он сравнивает временные метки своего запроса и чужого. Побеждает тот, чья метка меньше. Если чужой запрос победил, то процесс посылает сообщение «OK». Если у чужого запроса метка больше, то ответ не посылается, а чужой запрос запоминается.

Выход из критической секции

После выхода из секции он посылает сообщение «OK» всем процессам, запросы от которых он запомнил, а затем стирает все запомненные запросы.

Временная оценка

Каждый процесс, когда пытается зайти в критическую секцию, посылает N-1 запросов с integer значением, для этого потребуется (N-1)*100 + 4*(N-1) времени. Так, как процесов N то получится, что для прохождения всеми нужно N*((N-1)*100 + 4*(N-1)) времени.

Компиляция и запуск программы

mpic++ ricart_agrawala.cppmpiexec -n 4 ./a.out

Описание программы

Данный программный код реализует децентрализованный алгоритм с временными метками для прохождения всеми процессами критических секций (алгоритм Ricart-Agrawala).

Для межпроцессного взаимодействия использовались специальные команды

```
MPI_Isend(
    const void *buf,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm,
    MPI_Request *request)
```

Данная команда отправляет неблокирующее сообщение сообщение с тегом tag, содержащее $count \times sizeof(datatype)$ байт, процессу с уникальным индентификатором destination.

```
MPI_Irecv(
    void *buf,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Request *request)
```

Данная команда получает неблокирующее сообщение с тегом tag, содержащее $count \times sizeof(datatype)$ байт, от процесса с уникальным индентификатором source.

```
MPI_Iprobe(
    int source,
    int tag,
    MPI_Comm comm,
    int *flag,
    MPI_Status *status)
```

Данная команда, проверяет есть ли входящие неблокирующие сообщения с тегом tag, от процесса с уникальным индентификатором source. Если такого сообщения нету, то переменной flag присваивается значение 0.

Программа начинает своё выполнение в функции

```
int main(int argc, char **argv)
```

В ней каждой ветке присваивается временная логическая метка time, согласно номеру процесса, после чего процесс выполняет команду sleep(time)

В программе используются следующие вспомогательные функции

- `void check_incoming_request()`

  Функция определяет пришли ли неблокирующие запросы к ветке, когда она не запрашивала доступ к критической секции, либо уже покинула ее. В результате ее работы, ветка, которая отправила запрос, посылается ответ с разрешение на вход.

- `void send_request(int time)`

  Функция отправляет всем другим веткам, запрос на разрешение войти в критическую секцию.

- `int check_incoming_reply()`

  Функция проверяет, есть ли ответы с разрешение войти в критическую секцию. Возвращает количество полученных разрешений.

- `void critical_section()`
  `}`

  Функция реализующая критическую секцию.

- ```
  std::stack<int> check_incoming_request_while_requesting_access(
      int time,
      std::stack<int> stack_of_requests)
  ```

  Функция проверяющая, наличие входящие запросов к критической секции, во время того, как ветка сама запрашивает доступ к критической секции.Функция сравнивает временные метки своей ветки и чужой. Побеждает та, чья метка меньше. Если чужая ветка победила, то процесс посылает сообщение «OK». Если у чужой ветки метка больше, то ответ не посылается, а чужой запрос запоминается в stack_of_requests.
  ```

Код программы

```
#include "fstream"
#include <iostream>
#include "unistd.h"
#include "random"
#include <ctime>
#include <mpi/mpi.h>
#include "stack"
#define REPLY_MESSAGE 1
#define SEND_REQUEST 0

int size, rank;

void check_incoming_request(){
    int time, flag = 0;
    MPI_Request request;
    MPI_Status status;
    MPI_Iprobe(MPI_ANY_SOURCE, SEND_REQUEST, MPI_COMM_WORLD, &flag, &status);
    int reply = 1;
    while(flag){
        flag = 0;
        MPI_Irecv(&time, 1, MPI_INT,status.MPI_SOURCE,
                  SEND_REQUEST,MPI_COMM_WORLD,&request);
        printf("Rank: %d | Receive request from %d\n", rank, status.MPI_SOURCE);
        MPI_Isend(&reply,1,MPI_INT,status.MPI_SOURCE,REPLY_MESSAGE,
                  MPI_COMM_WORLD,&request);
        printf("Rank: %d | Send reply to %d\n", rank, status.MPI_SOURCE);
        MPI_Iprobe(MPI_ANY_SOURCE, SEND_REQUEST, MPI_COMM_WORLD, &flag, &status);
    }
    return;
}

void send_request(int time){
    int time_to_send = time;
    MPI_Request request;
    MPI_Status status;
    for (int i = 0; i < size; ++i){
        if (i != rank){
            MPI_Isend(&time_to_send,1,MPI_INT,i,
                      SEND_REQUEST,MPI_COMM_WORLD,&request);
            printf("Rank: %d | Send request to %d\n", rank, i);
        }
    }
}

int check_incoming_reply(){
    int time, flag = 0, answers = 0;
    MPI_Request request;
    MPI_Status status;
    MPI_Iprobe(MPI_ANY_SOURCE, REPLY_MESSAGE, MPI_COMM_WORLD, &flag, &status);
    int reply = 1;
```

```
51      while(flag){
52          flag = 0;
53          MPI_Irecv(&time, 1, MPI_INT,status.MPI_SOURCE,REPLY_MESSAGE,
54                    MPI_COMM_WORLD,&request);
55          printf("Rank: %d | Receive reply from %d\n", rank, status.
      MPI_SOURCE);
56          answers++;
57          MPI_Iprobe(MPI_ANY_SOURCE, REPLY_MESSAGE, MPI_COMM_WORLD, &flag,
       &status);
58      }
59      return answers;
60  }
61
62  void critical_section(){
63      std::ifstream my_file;
64      my_file.open("critical.txt");
65      if(my_file.is_open()){
66          std::cout << "file exists\n";
67          my_file.close();
68          return;
69      } else{
70          std::fopen("critical.txt", "w");
71          int time_to_sleep = 1 + random() % 10;
72          sleep(time_to_sleep);
73          std::remove("critical.txt");
74          std::cout << "file does not exists\n";
75          return;
76      }
77  }
78
79  std::stack<int> check_incoming_request_while_requesting_access(int time,
       std::stack<int> stack_of_requests){
80      int tmp_time, flag = 0;
81      MPI_Request request;
82      MPI_Status status;
83      MPI_Iprobe(MPI_ANY_SOURCE, SEND_REQUEST, MPI_COMM_WORLD, &flag, &
      status);
84
85      int reply = 1;
86      while(flag){
87          flag = 0;
88          MPI_Irecv(&tmp_time, 1, MPI_INT,status.MPI_SOURCE,
89                    SEND_REQUEST,MPI_COMM_WORLD,&request);
90          printf("Rank: %d | Receive request from %d\n", rank, status.
      MPI_SOURCE);
91          if (tmp_time > time){
92              printf("Rank: %d | Save request from %d\n", rank, status.
      MPI_SOURCE);
93              stack_of_requests.push(status.MPI_SOURCE);
94          }else{
95              MPI_Isend(&reply,1,MPI_INT,status.MPI_SOURCE,REPLY_MESSAGE,
96                        MPI_COMM_WORLD,&request);
97              printf("Rank: %d | Send reply to %d\n", rank, status.
      MPI_SOURCE);
98          }
99          MPI_Iprobe(MPI_ANY_SOURCE, SEND_REQUEST, MPI_COMM_WORLD, &flag,
      &status);
```

```
100     }
101     return stack_of_requests;
102 }
103
104 int main(int argc, char *argv[]){
105     MPI_Init( &argc, &argv );
106     int time, answers = 0;
107     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
108     MPI_Comm_size(MPI_COMM_WORLD, &size);
109     int* RD = (int *) malloc(size * sizeof(int));
110     for (int i = 0; i < size; ++i) {
111         RD[i] = 0;
112     }
113     std::stack<int> stack_of_requests;
114
115     time = rank;
116     printf("rank %d will sleep %d\n",rank, time);
117     sleep(time);
118     MPI_Request request;
119
120     //section with check before asking access to critical section
121     check_incoming_request();
122     //request access to critical section
123     send_request(time);
124     //waiting answer from other threads
125     while(answers != size-1){
126         printf("Rank: %d | Receive answers %d\n", rank, answers);
127         stack_of_requests =
      check_incoming_request_while_requesting_access(time,
      stack_of_requests);
128         answers += check_incoming_reply();
129         sleep(1);
130     }
131     printf("Rank: %d | ENTERING critical section \n", rank);
132     critical_section();
133     printf("Rank: %d | EXIT critical section \n", rank);
134
135     check_incoming_request();
136     while (!stack_of_requests.empty()){
137         printf("%d received message from %d\n",rank, stack_of_requests.
      top());
138         int message = 1;
139         MPI_Isend(&message, 1, MPI_INT, stack_of_requests.top(),
140                   REPLY_MESSAGE, MPI_COMM_WORLD, &request);
141         stack_of_requests.pop();
142     }
143     check_incoming_request();
144
145     MPI_Finalize();
146
147     return 1;
148 }
```

Листинг 1: ricart_agrawala.cpp

Выводы

В ходе работы был реализован алгоритм, использующий децентрализованный алгоритм с временными метками для прохождения всеми процессами критических секций. Реализация была выполнена в виде программы на C++ с использованием библоитеки MPI.