МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Практикум по учебному курсу
«Распределенные системы»

Централизованный алгоритм на транспьютерной матрице для прохождения всеми процессами критических секций

Разработка отказоустойчивой параллельной версии программы для задачи подсчета интеграла методом прямоугольников

Отчет
студента 420 группы
Трубецкого Сергея

Москва, 2022

Содержание

1 Постановка задачи

Требуется сделать следующее:

- 16 процессов, находящихся в узлах транспьютерной матрицы размером 4*4, одновременно выдали запрос на вход в критическую секцию. Реализовать программу, использующую централизованный алгоритм на транспьютерной матрице для прохождения всеми процессами критических секций. Получить временную оценку работы алгоритма. Оценить сколько времени потребуется для прохождения всеми критических секций, если координатор расположен в узле (0,0)? Время старта равно 100, время передачи байта равно 1 (Ts=100,Tb=1). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

- Доработать MPI-программу, реализованную в рамках курса “Суперкомпьютеры и параллельная обработка данных”. Добавить контрольные точки для продолжения работы программы в случае сбоя. Реализовать один из 3-х сценариев работы после сбоя:
  A. продолжить работу программы только на “исправных” процессах;
  B. вместо процессов, вышедших из строя, создать новые MPI-процессы, которые необходимо использовать для продолжения расчетов;
  C. при запуске программы на счет сразу запустить некоторое дополнительное количество MPI-процессов, которые использовать в случае сбоя.

- Подготовить отчет о выполнении задания, включающий описание алгоритма, детали реализации, а также временные оценки работы алгоритма.

Дополнительные комментарии:

- Критическая секция:

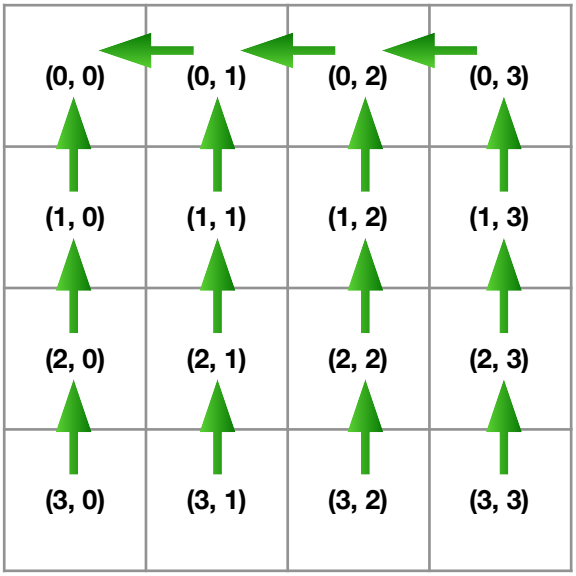
  <проверка наличия файла “critical.txt”>;
  if (<файл “critical.txt” существует>) {
  <сообщение об ошибке>;
  <завершение работы программы>;
  } else {
  <создание файла “critical.txt”>;
  Sleep (<случайное время>);
  <уничтожение файла “critical.txt”>;
  }
- Для межпроцессорных взаимодействий использовать средства MPI.

2 Реализация централизованного алгоритма и оценка его сложности

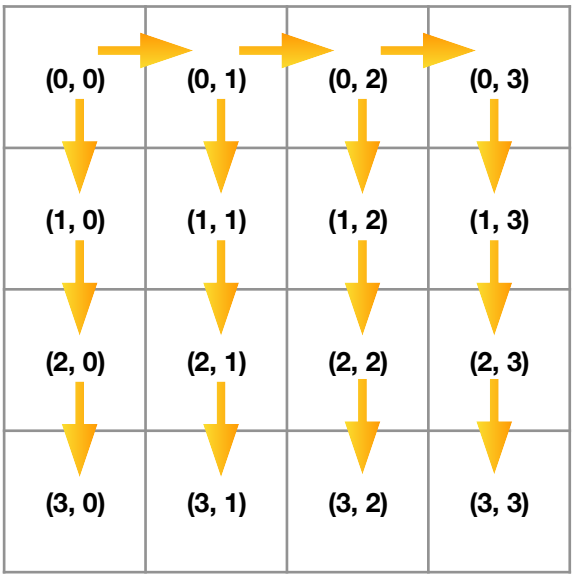
Что такое централизованный алгоритм:

Все процессы запрашивают у координатора разрешение на вход в критическую секцию и ждут этого разрешения. Координатор обслуживает запросы в порядке поступления. Получив разрешение процесс входит в критическую секцию. При выходе из нее он сообщает об этом координатору. Количество сообщений на одно прохождение критической секции = 3.

Нам нужно реализовать централизованный алгоритм на транспьютерной матрице, но мы не можем из любого процесса запросить у (0, 0) доступ к критической секции напрямую. Нужно передавать запрос последовательно через соседний процесс на транспьютерной матрице. Так же нужно поступать с разрешением на вход и с сообщением о выходе из критической секции. Один из способов пересылки (который был реализован) показан на рисунке:



а)



б)

Схема пересылки для: а) запроса разрешения на вход в критическую секцию, сообщения о выходе из критической секции; б) разрешения от координатора на вход в критическую

Данный алгоритм был реализован с помощью функций *MPI_Send* и *MPI_Recv*. Получение топологии в виде транспьютерной матрицы произведено с помощью функции *MPI_Cart_create*.

Оценим время работы алгоритма. Если время старта равно 100, время передачи байта равно 1 (Ts=100,Tb=1), то время выполнения операции рассчитывается следующим образом:

- Для обработки КС централизованным алгоритмом нужно 3 сообщения (запрос, разрешение, подтверждение окончания). Всего от всех процессов до (0,0) нужно совершить 48 переходов (2*1 + 3*2 + 4*3 + 3*4 + 2*5 + 6). Значит, всего потратим времени 3*48*(Ts+Tb) = 14544.

Код программы:

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#define MAX_SLEEP_TIME 10
const char *critical_file = "critical.txt";

#include "mpi.h"
#include "queue.h"

#define START_ENTER 0
#define FINISH_ENTER 1
#define ALLOW_ENTER 2

#define DIM 4

void critical_section(MPI_Comm comm, int rank)
{
    int fd;
    if (!access(critical_file, F_OK)) {
        printf("Ошибка: файл %s существовал во время обработки
критической секции %d\n", critical_file, rank);
        MPI_Abort(comm, MPI_ERR_FILE_EXISTS);
    } else {
        fd = open(critical_file, O_CREAT, S_IRWXU);
        if (!fd) {
            printf("Ошибка: невозможно создать файл %s процессом
%d\n", critical_file, rank);
            MPI_Abort(comm, MPI_ERR_FILE);
        }
        int time_to_sleep = random() % MAX_SLEEP_TIME;
        sleep(time_to_sleep);
```

```
        if (close(fd)) {
            printf("Ошибка: не удалось закрыть файл %s процессом
%d\n", critical_file, rank);
            MPI_Abort(comm, MPI_ERR_FILE);
        }
        if (remove(critical_file)) {
            printf("Ошибка: не удалось удалить файл %s процессом
%d\n", critical_file, rank);
            MPI_Abort(comm, MPI_ERR_FILE);
        }
    }
}

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    srand(time(NULL));
    MPI_Comm comm;
    int rank, send2master_rank, send2proc_rank, master_rank;
    int coords[2], send2master_coords[2], send2proc_coords[2],
master_coords[2] = {0};
    int buf;
    MPI_Status status;
    MPI_Request req;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // определяем ранк
процесса

    int dims[2] = {DIM, DIM};
    int periodic[2] = {0};
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periodic, 0, &comm);
// создание транспьютерной матрицы
    MPI_Cart_coords(comm, rank, 2, coords); // получаем
координаты текущего процесса в транспьютерной матрице
    MPI_Cart_rank(comm, master_coords, &master_rank); // получаем
rank координатора


    if (rank != master_rank) {
        int num_rcvs = ((coords[0] == 0) ? DIM * (DIM -
coords[1]) : DIM - coords[0]) - 1;
```

```
        if (coords[0] != 0) {
            send2master_coords[0] = coords[0] - 1;
            send2master_coords[1] = coords[1];
        } else {
            send2master_coords[0] = coords[0];
            send2master_coords[1] = coords[1] - 1;
        }
        MPI_Cart_rank(comm, send2master_coords,
&send2master_rank);
        buf = rank;
        MPI_Send(&buf, 1, MPI_INT, send2master_rank, START_ENTER,
comm);
        printf("Отправлен запрос на разрешение на вход для (%d,
%d)\n", coords[0], coords[1]);
        for (int i = 0; i < 3*num_rcvs + 1; i++) {
            MPI_Recv(&buf, 1, MPI_INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, comm, &status);
            if (status.MPI_TAG == START_ENTER) {
                MPI_Send(&buf, 1, MPI_INT, send2master_rank,
START_ENTER, comm);
            } else if (status.MPI_TAG == ALLOW_ENTER) {
                if (buf == rank) {
                    printf("Получено разрешение на вход для (%d,
%d)\n", coords[0], coords[1]);
                    critical_section(comm, rank);
                    MPI_Send(&buf, 1, MPI_INT, send2master_rank,
FINISH_ENTER, comm);
                } else {
                    MPI_Cart_coords(comm, buf, 2,
send2proc_coords);
                    if (send2proc_coords[1] > coords[1]) {
                        send2proc_coords[0] = coords[0];
                        send2proc_coords[1] = coords[1] + 1;
                    } else {
                        send2proc_coords[0] = coords[0] + 1;
                        send2proc_coords[1] = coords[1];
                    }
                    MPI_Cart_rank(comm, send2proc_coords,
&send2proc_rank);
```

```
                    MPI_Send(&buf, 1, MPI_INT, send2proc_rank,
ALLOW_ENTER, comm);
                }
            } else if (status.MPI_TAG == FINISH_ENTER) {
                MPI_Send(&buf, 1, MPI_INT, send2master_rank,
FINISH_ENTER, comm);
            }
        }
    } else { // coordinator actions
        for (int i = 0; i < 2 * SIZE; ++i) {
            MPI_Recv(&buf, 1, MPI_INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, comm, &status);
            if (status.MPI_TAG == START_ENTER) {
                if (is_free) {
                    is_free = 0;
                    MPI_Cart_coords(comm, buf, 2, coords);
                    printf("Разрешен вход для (%d, %d)\n",
coords[0], coords[1]);
                    if (coords[1] > 0) {
                        coords[0] = 0;
                        coords[1] = 1;
                    } else {
                        coords[0] = 1;
                        coords[1] = 0;
                    }
                    MPI_Cart_rank(comm, coords, &rank);
                    MPI_Send(&buf, 1, MPI_INT, rank, ALLOW_ENTER,
comm);
                } else {
                    enQueue(buf);
                }
            } else if (status.MPI_TAG == FINISH_ENTER) {
                is_free = 1;
                MPI_Cart_coords(comm, buf, 2, coords);
                printf("Завершена обработка критической секции
для (%d, %d)\n", coords[0], coords[1]);
                int next_proc = deQueue();
                if (next_proc != -1) {
                    is_free = 0;
```

```
                    MPI_Cart_coords(comm, next_proc, 2, coords);
                    printf("Разрешен вход для (%d, %d)\n",
coords[0], coords[1]);
                    if (coords[1] > 0) {
                        coords[0] = 0;
                        coords[1] = 1;
                    } else {
                        coords[0] = 1;
                        coords[1] = 0;
                    }
                    MPI_Cart_rank(comm, coords, &rank);
                    MPI_Isend(&next_proc, 1, MPI_INT, rank,
ALLOW_ENTER, comm, &req);
                }
            }
        }
        critical_section(comm, master_rank);
        printf("Завершена обработка критической секции для
координатора\n");
    }
    MPI_Finalize();
    return 0;
}
```

3 Добавление в программу возможности ее продолжения в случае сбоя

Для того, чтобы при сбое одного из процессов программа не завершалась с ошибкой, а продолжала своё выполнение, необходимо написать обработчик ошибок, который будет срабатывать в таких ситуациях. Для этого в стандарте MPI существует специальные функции *MPI_Comm_create_errhandler* и *MPI_Comm_set_errhandler*. Однако стандарт не позволяет определить, в каком именно процессе произошла ошибка. Это можно сделать, используя расширение MPI – ULFM.
Реализован сценарий:

а) продолжить работу программы только на “исправных” процессах;

Обработчик ошибок *verbose_errhandler* вычисляет процессы, вышедшие из строя. Корневой процесс пересчитывает участки данных, которые были предназначены для вышедших процессов.

Код программы:

```
#include <mpi.h>
#include <mpi-ext.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <math.h>

int* ranks_gc;
int nf = 0;
const float left_br = 0;  /* lower limit of integration */
const float right_br = 10000; /* upper limit of integration */

typedef double(*funcPtr)(double);

double func(double x) {
    return (1.0 / (1.0 + exp(-x)));
}

double integral(funcPtr f,double a, int num, double h) {
  double sum = 0;
  for(int i = 0; i < num; i += 1)
    sum += f(a + ((double)i + 0.5)*h) * h;
  return sum;
}
```

```c
static void verbose_errhandler(MPI_Comm* pcomm, int* perr, ...) {
    free(ranks_gc);
    MPI_Comm comm = *pcomm;
    int err = *perr;
    char errstr[MPI_MAX_ERROR_STRING];
    int i, rank, size, len, eclass;
    MPI_Group group_c, group_f;
    int *ranks_gf;

    MPI_Error_class(err, &eclass);
    if( MPIX_ERR_PROC_FAILED != eclass ) {
        MPI_Abort(comm, err);
    }

    MPIX_Comm_failure_ack(comm);
    MPIX_Comm_failure_get_acked(comm, &group_f);
    MPI_Group_size(group_f, &nf);

    ranks_gf = (int*)malloc(nf * sizeof(int));
    ranks_gc = (int*)malloc(nf * sizeof(int));
    MPI_Comm_group(comm, &group_c);
    for(i = 0; i < nf; i++)
        ranks_gf[i] = i;
    MPI_Group_translate_ranks(group_f, nf, ranks_gf,
                              group_c, ranks_gc);
    free(ranks_gf);
}

int main(int argc, char *argv[]) {
    int n, size, i, j, ierr,num;
    double h, result, a, b, pi;
    double my_a, my_range;
    double startwtime, my_time = 0.0, mintime = 0.0, time = 0.0;
    int rank, source, dest, tag, count;
    MPI_Status status;
    double my_result;
```

```
    n = 512;    /* number of increment within each process */
    dest = 0;         /* define the process that computes the
final result */
    tag = 123;        /* set the tag to identify this particular
job */
/* Starts MPI processes ... */
    // int rank, size;
    MPI_Errhandler errh;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_create_errhandler(verbose_errhandler,
                               &errh);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD,
                            errh);
    MPI_Barrier(MPI_COMM_WORLD);

    double* vec_r = (double*)malloc(size * sizeof(double));
    h = (right_br-left_br)/n;           /* length of increment */
    num = n / size;                       /* number of intervals
calculated by each process*/
    my_range = (right_br-left_br)/size;
    my_a = left_br + rank*my_range;

    startwtime = MPI_Wtime();
    my_result = integral(func,my_a,num,h);
    my_time = MPI_Wtime() - startwtime;

    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1)
     || rank == (size/2) ) {
        printf("Rank : %d, my_a: %f, my_res: %f\n", rank, my_a,
my_result);
        printf("Rank %d / %d: bye bye!\n\n", rank, size);
        raise(SIGKILL);
    }
    if(rank == 0) {
      vec_r[0] = my_result;
      time = my_time;
```

```
      for (int i=1; i<size; i++) {
        source = i;           /* MPI process number range is
[0,size-1] */
        MPI_Recv(&my_result, 1, MPI_DOUBLE, source, tag,
                      MPI_COMM_WORLD, &status);
        vec_r[i] = my_result;
        MPI_Recv(&my_time, 1, MPI_DOUBLE, source, tag-1,
                      MPI_COMM_WORLD, &status);
        time = fmax(time, my_time);
      }
    }
    else {
      MPI_Send(&my_result, 1, MPI_DOUBLE, dest, tag,
                    MPI_COMM_WORLD);      /* send my_result to
intended dest.*/
      MPI_Send(&my_time, 1, MPI_DOUBLE, dest, tag-1,
                    MPI_COMM_WORLD);      /* send my_time to
intended dest.*/
    }
    if(rank == 0) {
        if(nf != 0) {
            for(int i = 0; i < nf; i++) {
                my_a = left_br + ranks_gc[i] * my_range;

                startwtime = MPI_Wtime();
                my_result = integral(func,my_a,num,h);
                my_time = MPI_Wtime() - startwtime;

                printf("Recalculate : %d, my_a: %f, my_res:
%f\n", ranks_gc[i], my_a, my_result);
                vec_r[ranks_gc[i]] = my_result;
                time = fmax(time, my_time);
            }
            nf = 0;
        }
        result = 0;
        for(int i=0; i < size; i++) {
            result+=vec_r[i];
        }
        printf("\nRESULT: %f\n", result);
```

```
        printf("Time: %f\n", time);
    }
    free(vec_r);
    MPI_Finalize();
    return 0;
}
```

4 Заключение

Таким образом, был реализован централизованный алгоритм на транспьютерной матрице для прохождения всеми процессами критических секций при помощи пересылок MPI типа точка-точка и оценено время его работы. MPI-программа, реализующая подсчет интеграла методом прямоугольников, была доработана так, чтобы работа программы продолжалась после выхода из строя одного из процессов.