Московский Государственный Университет им. М.В. Ломоносова
Факультет Вычислительной Математики и Кибернетики
Кафедра Автоматизации Систем Вычислительных Комплексов

Задание по курсу "Распределённые системы"

Отчёт

Выполнил:
Савицкий Илья Павлович
421 группа

Москва, 2022 год

Задание 1

Формулировка

В транспьютерной матрице $6 * 6$, в каждом узле которой находится один процесс, необходимо выполнить операцию редукции MPI_MAXLOC, определить глобальный максимум и соответствующих ему индексов. Каждый процесс предоставляет свое значение и свой номер в группе. Для всех процессов операция редукции должна возвратить значение максимума и номер первого процесса с этим значением. Реализовать программу, моделирующую выполнение данной операции на транспьютерной матрице при помощи пересылок MPI типа точка-точка. Оценить сколько времени потребуется для выполнения операции редукции, если все процессы выдали эту операцию редукции одновременно. Время старта равно 100, время передачи байта равно 1 ($T_s = 100$, $T_b = 1$). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

Решение

В рамках решения был разработан следующий алгоритм, состоящий из 6 этапов:

Алгоритм

1. Для каждого из крайних строк транспьютерной матрицы сравнить значений конкретного ряда и внутреннего ряда. Тогда в 1 и 4 строках будут находится максимумы с 4 строк
2. Проделать то же самое для двух внутренних строк 2 и 3
3. Проделать то же самое, но схлопнуть все в строку 2
4. Начать схлопывать края - 0 и 5 позиции в 1 и 4
5. Аналогично предыдущему пункту, но 1 и 4 в 2 и 3
6. Схлопнуть позиции 2 и 3 в 2 - тогда минимум и координаты находятся в 2 строке во втором ряду
7. Переслать результат из 2,2 в нужный процессор

Видно, что первые два этапа, а так же этапы 4 и 5 имеют схожую структуру, а, значит, их сожно выделить в две функции.

1. `void collide_rows()` - для схлопывания двух крайних строк в строки, ближние к центру.
2. `void compress_row()` - для схлопывания двух крайних столбцов в столбцы, ближние к центру.

Оценка сложности алгоритма

Сложность алгоритма оценивается формулой

$$time = num\_steps \cdot (T_s + n \cdot T_b)$$

где num_steps - количество шагов алгоритма, n - размер сообщения, T_s - время старта, T_b - время передачи байта. В нашем случае $num\_steps = 6$, $T_s = 100$, $T_b = 1$. Размер сообщения (при предположении что `sizeof(int) == 4`) равен 12, так как помимо самого числа надо передать и координаты, что два числа. Тогда

$$time = 7 \cdot (100 + 12 \cdot 1) = 700 + 84 = 784$$

Текст программы

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

#define SIZE 6

void report(const char *msg, int coords[2]) {
    if (coords[0] == 0 && coords[1] == 0) {
        printf("(0,0) %s!\n", msg);
    }
}

void send_coords_and_value(int coords[2], int value,
↪   int other_rank,
                           MPI_Comm comm) {
    MPI_Send(coords, 2, MPI_INT, other_rank, 0,
↪   comm);
```

```
    MPI_Send(&value, 1, MPI_INT, other_rank, 0,
 ↪  comm);
}

void receive_coords_and_value(int *value, int
 ↪  other_coords[2], int other_rank,
                               MPI_Comm comm) {
    MPI_Recv(other_coords, 2, MPI_INT, other_rank,
 ↪  0, comm, MPI_STATUS_IGNORE);
    MPI_Recv(value, 1, MPI_INT, other_rank, 0, comm,
 ↪  MPI_STATUS_IGNORE);
}

void collide_rows(int row_1, int row_2, int
 ↪  coords[2], int *a,
                  int best_coords[2], MPI_Comm comm)
 ↪  {
    int result = 0;
```

```
    int recieved_coords[2];
    int other_rank = 0;
    int other_coords[2];
    other_coords[1] = coords[1];
    if (coords[0] == row_1 || coords[0] == row_2) {
        other_coords[0] = coords[0] == row_1 ?
↪   coords[0] + 1 : coords[0] - 1;
        MPI_Cart_rank(comm, other_coords,
↪   &other_rank);
        send_coords_and_value(best_coords, *a,
↪   other_rank, comm);
    } else if (coords[0] == row_1 + 1 || coords[0]
↪   == row_2 - 1) {
        other_coords[0] =
            coords[0] == row_1 + 1 ? coords[0] - 1 :
↪   coords[0] + 1;
        MPI_Cart_rank(comm, other_coords,
↪   &other_rank);
```

```
        receive_coords_and_value(&result,
 ↪  recieved_coords, other_rank, comm);
        if (result > *a) {
            *a = result;
            best_coords[0] = recieved_coords[0];
            best_coords[1] = recieved_coords[1];
        }
    }
}

void compress_row(int row_n, int pl_l, int pl_r, int
 ↪  coords[2], int *a,
                  int best_coords[2], MPI_Comm comm)
 ↪  {
    int result = 0;
    int recieved_coords[2];
    int other_rank = 0;
    int other_coords[2];
```

```
    other_coords[0] = coords[0];


    if (coords[0] == row_n && (coords[1] == pl_l ||
↪   coords[1] == pl_r)) {
        other_coords[1] = coords[1] == pl_l ?
↪   coords[1] + 1 : coords[1] - 1;
        MPI_Cart_rank(comm, other_coords,
↪   &other_rank);
        send_coords_and_value(best_coords, *a,
↪   other_rank, comm);
    }
    if (coords[0] == row_n &&
        (coords[1] == pl_l + 1 || coords[1] == pl_r
↪   - 1)) {
        other_coords[1] = coords[1] == pl_l + 1 ?
↪   coords[1] - 1 : coords[1] + 1;
```

```
            MPI_Cart_rank(comm, other_coords,
    ↪   &other_rank);
            receive_coords_and_value(&result,
    ↪   recieved_coords, other_rank, comm);
            if (result > *a) {
                *a = result;
                best_coords[0] = recieved_coords[0];
                best_coords[1] = recieved_coords[1];
            }
        }
    }

    int main(int argc, char *argv[]) {
        MPI_Init(&argc, &argv);
        int rank, tasks;
        MPI_Comm comm;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &tasks);
```

```
    int size[2] = {SIZE, SIZE};
    int periodic[2] = {0};
    MPI_Cart_create(MPI_COMM_WORLD, 2, size,
↪   periodic, 0, &comm);
    int coords[2];
    MPI_Cart_coords(comm, rank, 2, coords);
    srand(rank + 6);
    int a = rand() % 1000;
    printf("Coordinates for process %d: (%d, %d)\n",
↪   rank, coords[0],
           coords[1]);
    printf("a[%d][%d] = %d\n", coords[0], coords[1],
↪   a);


    int result = 0;
    int other_coords[2];
    int recieved_coords[2];
```

```
    int best_coords[2];
    best_coords[0] = coords[0];
    best_coords[1] = coords[1];
    int other_rank = 0;

    other_coords[1] = coords[1];
    collide_rows(0, 5, coords, &a, best_coords,
↪   comm);
    MPI_Barrier(comm);
    report("step 1", coords);

    collide_rows(1, 4, coords, &a, best_coords,
↪   comm);
    MPI_Barrier(comm);
    report("step 2", coords);

    if (coords[0] == 3) {
        other_coords[0] = coords[0] - 1;
```

```
        MPI_Cart_rank(comm, other_coords,
↪   &other_rank);
        send_coords_and_value(best_coords, a,
↪   other_rank, comm);
    }
    if (coords[0] == 2) {
        other_coords[0] = coords[0] + 1;
        MPI_Cart_rank(comm, other_coords,
↪   &other_rank);
        receive_coords_and_value(&result,
↪   recieved_coords, other_rank, comm);
        if (result > a) {
            a = result;
            best_coords[0] = recieved_coords[0];
            best_coords[1] = recieved_coords[1];
        }
    }
    MPI_Barrier(comm);
```

```
    report("step 3", coords);

    compress_row(2, 0, 5, coords, &a, best_coords,
↪   comm);
    MPI_Barrier(comm);
    report("step 4", coords);

    compress_row(2, 1, 4, coords, &a, best_coords,
↪   comm);
    MPI_Barrier(comm);
    report("step 5", coords);

    if (coords[0] == 2 && coords[1] == 3) {
        other_coords[0] = coords[0];
        other_coords[1] = 2;
        MPI_Cart_rank(comm, other_coords,
↪   &other_rank);
```

```
        send_coords_and_value(best_coords, a,
↪   other_rank, comm);
    }
    if (coords[0] == 2 && coords[1] == 2) {
        other_coords[0] = coords[0];
        other_coords[1] = 3;
        MPI_Cart_rank(comm, other_coords,
↪   &other_rank);
        receive_coords_and_value(&result,
↪   recieved_coords, other_rank, comm);
        if (result > a) {
            a = result;
            best_coords[0] = recieved_coords[0];
            best_coords[1] = recieved_coords[1];
        }
    }
    MPI_Barrier(comm);
    report("step 6", coords);
```

```
    if (coords[0] == 2 && coords[1] == 2) {
        printf("Max result: %d on (%d, %d)\n", a,
↪   best_coords[0],
               best_coords[1]);
    }
    MPI_Finalize();
    return 0;
}
```

Задание 2

Формулировка

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:

1. продолжить работу программы только на “исправных”

процессах;

2. вместо процессов, вышедших из строя, создать новые MPI-процессы, которые необходимо использовать для продолжения расчетов;

3. при запуске программы на счет сразу запустить некоторое дополнительное количество MPI-процессов, которые использовать в случае сбоя.

Решение

```
#include <math.h>
#include <mpi-ext.h>
#include <mpi.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.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 (*math_func)(double);

double integrate(math_func f, double a, int num,
 ↪  double h) {
    int myid, numprocs;
    double x, sum = 0.0;
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    for (int i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
```

```
        sum += f(x);
    }
    return h * sum;
}

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);
}

MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);

MPIX_Comm_failure_ack(comm);
MPIX_Comm_failure_get_acked(comm, &group_f);
MPI_Group_size(group_f, &nf);
MPI_Error_string(err, errstr, &len);

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;
    a = 0.;
    b = 1;
    n = (argc > 1) ? pow(2, atoi(argv[1]))
                   : 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 = integrate(sqrt, 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 = integrate(sqrt, 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;
}
```