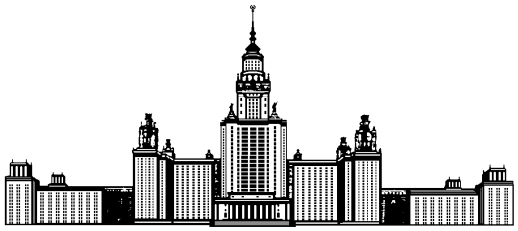
Московский государственный университет имени М. В. Ломоносова



Факультет Вычислительной Математики и Кибернетики
Кафедра Системного программирования

Отчет по заданию курса "Распределенные системы"

"Разработка программы, использующей древовидный маркерный алгоритм для прохождения всеми процессами критических секций"

"Отказоустойчивая параллельная версия программы для сортировки данных"

Выполнил:
студент 4 курса 427 группы
Майоров Егор Андреевич

Москва, 2023

Содержание

1 Задание 1

1.1 Постановка задачи

Все 16 процессов, находящихся на разных ЭВМ сети, одновременно выдали запрос на вход в критическую секцию. Реализовать программу, использующую древовидный маркерный алгоритм 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 передачи маркера использовать средства MPI.

Получить временную оценку работы алгоритма. Оценить сколько времени потребуется, если маркером владеет нулевой процесс. Время старта (время «разгона» после получения доступа к шине для передачи сообщения) равно 100, время передачи байта равно 1 (Ts=100, Tb=1). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

1.2 Алгоритм

Древовидный маркерный алгоритм *Raymond*.

Все процессы представлены в виде сбалансированного двоичного дерева. Каждый процесс имеет очередь запросов от себя и соседних процессов и указатель в направлении владельца маркера.

Вход в критическую секцию:

- Если есть маркер, то процесс выполняет критическую секцию.
- Если нет маркера, то процесс:
    1. Помещает свой запрос в очередь запросов
    2. Посылает сообщение $REQUEST$ в направлении маркера и ждет сообщений.

Поведение процесса при приеме сообщений:

Процесс, не находяшийся внутри критической секции должен реагировать на сообщения двух видов - $MARKER$ и $REQUEST$.

А) Пришло сообщение $MARKER$:

1. Взять первый запрос из очереди и послать маркер его автору (возможно себе);
2. Поменять значение указателя в сторону маркера;
3. Исключить запрос из очереди;
4. Если в очереди остались запросы, то послать сообщение $REQUEST$ в сторону маркера.

Б) Пришло сообщение $REQUEST$:

1. Поместить запрос в очередь;
2. Если нет маркера, то послать сообщение $REQUEST$ в сторону маркера, иначе если есть маркер - перейти на пункт А1.

Выход из критической секции.

Если очередь запросов пуста, то при выходе ничего не делается, иначе - перейти на пункт А1.

1.3 Реализация

Для реализации алгоритма была использована структура, используемая каждым процессом, которая содержит информацию о номере процесса, о его корне в двоичном дереве и о его правом и левом поддеревьях. Также структура хранит информацию о наличии маркера и указатель на маркер - $PARENT$, $LEFT$ или $RIGHT$.

Двоичное дерево строится следующим образом:

1. Корень дерева - процесс 0.
2. Номер левого листа - $rank * 2 + 1$ (либо -1).
3. Номер правого листа - $rank * 2 + 1$ (либо -1).
4. Номер родителя в дереве - $(rank - 1)/2$ (у процесса с номером 0 родителя нет, поэтому поле равно -1).

Код программы представлен в главе 3.1.

1.4 Временная оценка

По условию задачи мы имеем 16 процессов, которые образуют сбалансированное дерево. При этом в моей реализации процессы получают доступ к критической секции по очереди в порядке возрастания своего номера.

Будем считать, что маркер находится у процесса, лежащего на глубине k, корень дерева лежит на глубине 0. В таком случае сложность алгоритма будет равна $T_s + (128 + 2 * k) * T_b$. При подсчете сложности учитывались операции запроса маркера процессом и передачи маркера процессу, запросившего его.

Можно начать обход дерева с корня, тогда k будет равно нулю и время выполнения будет минимальным: $100 + (128 + 2 * 0) * 1 = 228$.

2 Задание 2

2.1 Постановка задачи

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.

2.2 Реализация

Для продолжения работы программы после сбоя был использован сценарий номер 3: при запуске программы на счет сразу запускается некоторое дополнительное количество MPI-процессов, которые используются в случае сбоя.

Для того, чтобы при сбое одного из процессов программа не завершалась с ошибкой, а продолжала свое выполнение, необходим обработчик ошибок. Для этого в стандарте MPI существуют $MPI\_Comm\_create\_errhandler$ и $MPI\_Comm\_set\_errhandler$. Для определения процесса, в котором произошла ошибка, используется расширение MPI $-$ $ULFM$.

Программа была доработана следующим образом:

1. С использованием функций $MPI\_Comm\_create\_errhandler$ и $MPI\_Comm\_set\_errhandler$ добавлен обработчик ошибок $err\_handler$.
2. В качестве резервного процесса используется последний процесс, который изначально не получает данных для обработки.
3. Для каждого работающего процесса создается файл для записи данных в контрольных точках.
4. В начале данные распределяются по процессам с помощью операции $MPI\_Scatterv$, при этом последнему процессу данные не отправляются.
5. Далее следует $tasks - 1$ итераций цикла обмена данными между процессами, где $tasks$ - число процессов, а $tasks - 1$ - число процессов, которые участвуют в обработке данных (без резервного).
6. В начале каждой итерации процессы читают данные из файла, работают с ними, и в конце итерации записывают данные обратно в файлы. Если произошла ошибка, то процессы начинают итерацию заново.
7. Процессы должны находиться на одних и тех же итерациях, для этого используется функция $MPI\_Barrier$.
8. В коде один из процессов убивается, после чего во всех процессах управление переходит в функцию $err\_handler$ - обработчик ошибок.
9. В обработчике ошибок создается новый коммуникатор, который не включает в себя вышедшие из строя процессы (то есть один процесс, который был убит). Для этого используется функция $MPIX\_Comm\_shrink$, которая не входит в стандарт MPI.
10. После создания нового коммуникатора каждый процесс получает новый номер. Возможно процессы будут работать с другими файлами, но это не влияет на выполнение программы.
11. Работа продолжается на оставшихся процессах и результат собирается в процессе с номером 0 с помощью функции MPI_Gather.

Код программы представлен в главе 3.2.

3 Code

3.1 task_1.cpp

```cpp
#include <iostream>
#include <queue>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <map>
#include <mpi.h>

#define TREE_ROOT 0

enum message_type : int8_t {REQUEST, MARKER, NOTHING};
enum pointer : int8_t {PARENT, LEFT, RIGHT};

struct tree {
    std::queue<int> request_queue;
    int rank;
    int root;
    int left;
    int right;
    pointer marker_pointer;
    bool marker;

    tree () {}

    void critical () {
        if (access("critical.txt", F_OK) != -1) {
            std::cerr << "File exist" << std::endl;
            MPI_Finalize();
            exit(1);
        } else {
            fopen("critical.txt", "w");
            std::cout << "CRITICAL SECTION - rank " << rank << std::endl;
            sleep(rand() % 10);
            remove("critical.txt");
        }
    }

    void request (message_type message, int reciever) {
        if (message == MARKER) {
            if (!request_queue.empty()) {
                int requester = request_queue.front();
                request_queue.pop();
                std::cout << "send marker from " << rank << " to " << requester << std::endl;
                if (requester == rank) {
                    marker = true;
                    critical();
                } else if (requester == root) {
                    marker = false;
                    marker_pointer = PARENT;
                    message_type tmp = MARKER;
                    MPI_Send(&tmp, 1, MPI_INT8_T, requester, 0, MPI_COMM_WORLD);
                } else if (requester == left) {
                    marker = false;
```

```
                    marker_pointer = LEFT;
                    message_type tmp = MARKER;
                    MPI_Send(&tmp, 1, MPI_INT8_T, requester, 0, MPI_COMM_WORLD);
                } else if (requester == right) {
                    marker = false;
                    marker_pointer = RIGHT;
                    message_type tmp = MARKER;
                    MPI_Send(&tmp, 1, MPI_INT8_T, requester, 0, MPI_COMM_WORLD);
                } else {
                    std::cerr << "Invalid rank in queue!" << std::endl;
                    MPI_Finalize();
                    exit(1);
                }
                if (!request_queue.empty()) {
                    int receiver = request_queue.front();
                    request_queue.pop();
                    request(REQUEST, receiver);
                }
            }
        } else if (message == REQUEST) {
            std::cout << "request from " << reciever << " to " << rank << std::endl;
            request_queue.push(reciever);
            if (marker) {
                request(MARKER, rank);
            } else {
                if (marker_pointer == PARENT) {
                    message_type tmp = REQUEST;
                    if (rank != 0)
                        MPI_Send(&tmp, 1, MPI_INT8_T, root, 0, MPI_COMM_WORLD);
                } else if (marker_pointer == LEFT) {
                    message_type tmp = REQUEST;
                    MPI_Send(&tmp, 1, MPI_INT8_T, left, 0, MPI_COMM_WORLD);
                } else if (marker_pointer == RIGHT) {
                    message_type tmp = REQUEST;
                    MPI_Send(&tmp, 1, MPI_INT8_T, right, 0, MPI_COMM_WORLD);
                } else {
                    std::cerr << "Invalid marker pointer" << std::endl;
                    MPI_Finalize();
                    exit(1);
                }
            }
        }
    }

    void print_info () {
        std::cout << "rank: " << rank << ", root: " << root << ", left: " << left
              << ", right: " << right;
        if (marker_pointer == PARENT){
            std::cout << ", marker pointer: PARENT";
        } else if (marker_pointer == LEFT) {
            std::cout << ", marker pointer: LEFT";
        } else if (marker_pointer == RIGHT) {
            std::cout << ", marker pointer: RIGHT";
        }
        std::cout << ", marker: " << marker << std::endl;
    }
```

```
};

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    int tasks, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &tasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int marker_node = atoi(argv[1]) % tasks;
    int marker_cnt = marker_node;
    std::map<int, int> from_root_to_marker;

    while (marker_cnt != TREE_ROOT) {
        from_root_to_marker[(marker_cnt - 1) / 2] = marker_cnt;
        marker_cnt = (marker_cnt - 1) / 2;
    }
    // создание структуры-дерева для каждого процесса
    tree tree_elem;
    if (rank == 0) {
        tree_elem.rank = TREE_ROOT;
        tree_elem.root = -1;
        tree_elem.left = 1;
        tree_elem.right = 2;
        tree_elem.marker_pointer = PARENT;
        tree_elem.marker = marker_node == TREE_ROOT;
    } else {
        int left = 2 * rank + 1;
        if (left >= tasks)
            left = -1;
        int right = 2 * rank + 2;
        if (right >= tasks)
            right = -1;
        tree_elem.rank = rank;
        tree_elem.root = (rank - 1) / 2;
        tree_elem.left = left;
        tree_elem.right = right;
        tree_elem.marker_pointer = PARENT;
        tree_elem.marker = marker_node == rank;
    }
    // заполнение указателей на маркер
    if (from_root_to_marker.count(rank)) {
        int relative_marker_path = from_root_to_marker[rank];
        if (tree_elem.left != -1 && relative_marker_path == tree_elem.left) {
            tree_elem.marker_pointer = LEFT;
        } else if (tree_elem.right != -1 && relative_marker_path == tree_elem.right) {
            tree_elem.marker_pointer = RIGHT;
        }
    }
    // полученное дерево
    tree_elem.print_info();
    // обмен сообщениями и получение маркера
    MPI_Barrier(MPI_COMM_WORLD);
    for (int request_sender = 0; request_sender < tasks; request_sender++) {
        if (rank == request_sender) {
            tree_elem.request(REQUEST, rank);
            message_type tmp_m;
            if (tree_elem.marker_pointer == PARENT) {
```

```
166                 if (rank != 0)
167                     MPI_Recv(&tmp_m, 1, MPI_INT8_T, tree_elem.root, 0, MPI_COMM_WORLD,
168                             MPI_STATUS_IGNORE);
169             } else if (tree_elem.marker_pointer == LEFT) {
170                 MPI_Recv(&tmp_m, 1, MPI_INT8_T, tree_elem.left, 0, MPI_COMM_WORLD,
171                         MPI_STATUS_IGNORE);
172             } else if (tree_elem.marker_pointer == RIGHT) {
173                 MPI_Recv(&tmp_m, 1, MPI_INT8_T, tree_elem.right, 0, MPI_COMM_WORLD,
174                         MPI_STATUS_IGNORE);
175             }
176             if (tmp_m == MARKER) {
177                 tree_elem.request(MARKER, tree_elem.root);
178             } else if (tmp_m == REQUEST) {
179                 tree_elem.request(REQUEST, tree_elem.root);
180             }
181             for (int i = 0; i < tasks; i++) {
182                 if (i != request_sender) {
183                     message_type tmp = NOTHING;
184                     MPI_Send(&tmp, 1, MPI_INT8_T, i, 0, MPI_COMM_WORLD);
185                 }
186             }
187         } else {
188             message_type inp;
189             do {
190                 MPI_Status status;
191                 MPI_Recv(&inp, 1, MPI_INT8_T, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
192                 if (inp == MARKER) {
193                     tree_elem.request(MARKER, status.MPI_SOURCE);
194                 } else if (inp == REQUEST) {
195                     tree_elem.request(REQUEST, status.MPI_SOURCE);
196                 }
197             } while (inp != NOTHING);
198         }
199         MPI_Barrier(MPI_COMM_WORLD);
200     }
201     MPI_Finalize();
202     return 0;
203 }
```

3.2 task_2.c

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

#define KILLED_PROCESS 1

int rank, tasks;
char filename[10];
unsigned error_occured = 0;
MPI_Comm main_comm;

void itoa (int n, char s[]) {
    int i = 0;
    do {
        s[i++] = n % 10 + '0';
    } while ((n /= 10) > 0);
    s[i] = '\0';
    int j, k;
    char c;
    for (j = 0, k = strlen(s)-1; j < k; j++, k--) {
        c = s[j];
        s[j] = s[k];
        s[k] = c;
    }
}

static void err_handler (MPI_Comm *pcomm, int *perr, ...) {
    error_occured = 1;
    int err = *perr;
    char errstr[MPI_MAX_ERROR_STRING];
    int size, nf, len;
    MPI_Group group_f;
    MPI_Comm_size(main_comm, &size);
    MPIX_Comm_failure_ack(main_comm);
    MPIX_Comm_failure_get_acked(main_comm, &group_f);
    MPI_Group_size(group_f, &nf);
    MPI_Error_string(err, errstr, &len);
    printf("Process with rank %d know about an error %s.\n", rank, errstr);
    // создаем новый коммуникатор без вышедшего из строя процесса
    MPIX_Comm_shrink(main_comm, &main_comm);
    MPI_Comm_rank(main_comm, &rank);
    itoa(rank, filename);
    strcat(filename, ".txt");
}

void print_array_to_file (int *a, int len_a, char *filename) {
    FILE *fp = fopen(filename, "w");
    for (int i = 0; i < len_a; ++i) {
        fprintf(fp, "%d ", a[i]);
    }
    fclose(fp);
```

```
56  }
57
58  void read_array_from_file (int *a, int len_a, char *filename) {
59      FILE *fp = fopen(filename, "r");
60      for (int i = 0; i < len_a; ++i) {
61          fscanf(fp, "%d", &a[i]);
62      }
63      fclose(fp);
64  }
65
66  void swap (int *x, int *y) {
67      int tmp = *x;
68      *x = *y;
69      *y = tmp;
70  }
71
72  void sort (int size, int* arr) {
73      int i, j;
74      for (i = size - 2; i >= 0; i--)
75          for (j = 0; j <= i; j++)
76              if (arr[j] > arr[j + 1])
77                  swap(&arr[j], &arr[j + 1]);
78  }
79
80  void merge (const int *in_a, int len_a, const int *in_b, int len_b, int *out) {
81      int i, j;
82      int out_count = 0;
83      for (i = 0, j = 0; i < len_a; i++) {
84          while ((in_b[j] < in_a[i]) && j < len_b) {
85              out[out_count++] = in_b[j++];
86          }
87          out[out_count++] = in_a[i];
88      }
89      while (j < len_b) {
90          out[out_count++] = in_b[j++];
91      }
92  }
93
94  void pairwise_exchange (int local_n, int *local_a, int send_rank, int recv_rank) {
95      // процесс-отправитель отправляет свой массив и ждет результата
96      // процесс-получатель сортирует массивы и возвращает нужную половину
97      int remote[local_n];
98      int buf_all[2 * local_n];
99      const int merge_tag = 1;
100     const int sorted_tag = 2;
101     if (rank == send_rank) {
102         MPI_Send(local_a, local_n, MPI_INT, recv_rank, merge_tag, main_comm);
103         if (error_occured == 1) {
104             return;
105         }
106         MPI_Recv(local_a, local_n, MPI_INT, recv_rank, sorted_tag, main_comm,
107                  MPI_STATUS_IGNORE);
108     } else {
109         MPI_Recv(remote, local_n, MPI_INT, send_rank, merge_tag, main_comm,
110                  MPI_STATUS_IGNORE);
111         if (error_occured == 1) {
```

```
112             return;
113         }
114         merge(local_a, local_n, remote, local_n, buf_all);
115         int their_start = 0, my_start = local_n;
116         if (send_rank > rank) {
117             their_start = local_n;
118             my_start = 0;
119         }
120         MPI_Send(&(buf_all[their_start]), local_n, MPI_INT, send_rank, sorted_tag,
121                  main_comm);
122         for (int i = my_start; i < my_start + local_n; i++)
123             local_a[i - my_start] = buf_all[i];
124     }
125 }
126
127 void parallel_odd_even_sort (int n, int *a) {
128     // предполагается, что размер массива делится нацело на число процессов
129     int *local_a = malloc(n / (tasks - 1) * sizeof(int));
130     int *sendcounts = malloc(tasks * sizeof(int));
131     int *displs = malloc(tasks * sizeof(int));
132     displs[0] = 0;
133     sendcounts[0] = n / (tasks - 1);
134     for (int i = 1; i < tasks; i++) {
135         displs[i] = displs[i - 1] + n / (tasks - 1);
136         sendcounts[i] = n / (tasks - 1);
137     }
138     sendcounts[tasks - 1] = 0;
139     // распределение массива по процессам, последнему процессу ничего не даем
140     if (rank == tasks - 1) {
141         MPI_Scatterv(a, sendcounts, displs, MPI_INT, local_a, 0, MPI_INT, 0,
142                      main_comm);
143     } else {
144         MPI_Scatterv(a, sendcounts, displs, MPI_INT, local_a, n / (tasks - 1),
145                      MPI_INT, 0, main_comm);
146         // сортируем часть массива, которая досталась процессу
147         sort(n / (tasks - 1), local_a);
148         print_array_to_file(local_a, n / (tasks - 1), filename);
149     }
150     // убиваем один из процессов
151     MPI_Barrier(main_comm);
152     if (rank == KILLED_PROCESS) {
153         raise(SIGKILL);
154     }
155     MPI_Barrier(main_comm);
156     // четно-нечетная сортировка
157     for (int i = 1; i <= tasks - 1; i++) {
158         // если случилась ошибка в каком-то процессе,
159         // то остальные процессы заново начинают итерацию
160         checkpoint:
161         MPI_Barrier(main_comm);
162         read_array_from_file(local_a, n / (tasks - 1), filename);
163         if (error_occured == 1) {
164             error_occured = 0;
165             goto checkpoint;
166         }
167         // у номера процесса и номера итерации одинаковая четность
```

```
168             if ((i + rank) % 2 == 0) {
169                 if (rank < tasks - 2) {
170                     pairwise_exchange(n / (tasks - 1), local_a, rank, rank + 1);
171                     if (error_occured == 1) {
172                         error_occured = 0;
173                         goto checkpoint;
174                     }
175                 }
176             } else if (rank > 0) {
177                 pairwise_exchange(n / (tasks - 1), local_a, rank - 1, rank);
178                 if (error_occured == 1) {
179                     error_occured = 0;
180                     goto checkpoint;
181                 }
182             }
183             MPI_Barrier(main_comm);
184             print_array_to_file(local_a, n / (tasks - 1), filename);
185         }
186         MPI_Barrier(main_comm);
187         // собираем части массива в один
188         MPI_Gather(local_a, n / (tasks - 1), MPI_INT, a, n / (tasks - 1), MPI_INT, 0,
189                    main_comm);
190         free(local_a);
191     }
192
193     // При запуске программы нужно сразу запустить на один MPI-процесс больше,
194     // на случай, если один из процессов откажет,
195     // то дополнительный процесс начнет работу в случае сбоя.
196     int main (int argc, char **argv) {
197         MPI_Init(&argc, &argv);
198         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
199         MPI_Comm_size(MPI_COMM_WORLD, &tasks);
200         main_comm = MPI_COMM_WORLD;
201         // устанавливаем обработчик ошибок
202         MPI_Errhandler errh;
203         MPI_Comm_create_errhandler(err_handler, &errh);
204         MPI_Comm_set_errhandler(main_comm, errh);
205         MPI_Barrier(main_comm);
206         // для каждого процесса формируем имя файла для записи данных контрольных точек
207         itoa(rank, filename);
208         strcat(filename, ".txt");
209         int size = atoi(argv[1]);
210         int *a = NULL;
211         if (rank == 0) {
212             a = malloc(size * sizeof(int));
213             // Заполнение массива случайными числами
214             for (int i = 0; i < size; i++)
215                 a[i] = rand() % size;
216             double start_time = MPI_Wtime();
217             parallel_odd_even_sort(size, a);
218             double end_time = MPI_Wtime();
219             printf("\nParallel time: %f\n", end_time - start_time);
220             free(a);
221         } else {
222             parallel_odd_even_sort(size, a);
223         }
```

```
224         MPI_Barrier(main_comm);
225         remove(filename);
226         MPI_Finalize();
227         return 0;
228     }
```