Московский государственный университет
имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики

Задание по курсу

«Распределённые системы»

Реализация работы функции MPI_ALLTOALL в транспьютерной матрице размера 5x5 с использованием пересылок MPI типа точка-точка.

Отчёт
О выполненном задании

Выполнил:
студент 423 группы
Латыпов Ш. И.

Москва
2022

Содержание

Описание задачи

Задача состояла в то, чтобы реализовать программу, моделирующую выполнение операции MPI_ALLTOALL на транспьютерной матрице при помощи пересылок MPI типа точка-точка. Также необходимо получить временную оценку работы алгоритма. Оценить сколько времени потребуется для выполнения операции MPI_ALLTOALL, если все процессы выдали ее одновременно.

По условию была дана транспьютерная матрица размером 5*5, в каждом узле которой находился один процесс. Каждый процесс рассылает различные данные каждому получателю (массив из 25 целочисленных значений). j-й элемент массива, посланный процессом i, принимается процессом j и помещается в i-й элемент результирующего буфера.

Алгоритм программы

Первым шагом программы выполняли пересылку своих буферов вдоль строк. Для этого реализован цикл *while*, который итерировался по двум условиям - значениям левой и правой границ. Эти обозначения границ были необходимы, чтобы обозначать, какие из процессов должны на каждом шаге выполнять только операцию MPI_Send, MPI_Recv или MPI_Sendrecv_replace для избежаниия дедлока.

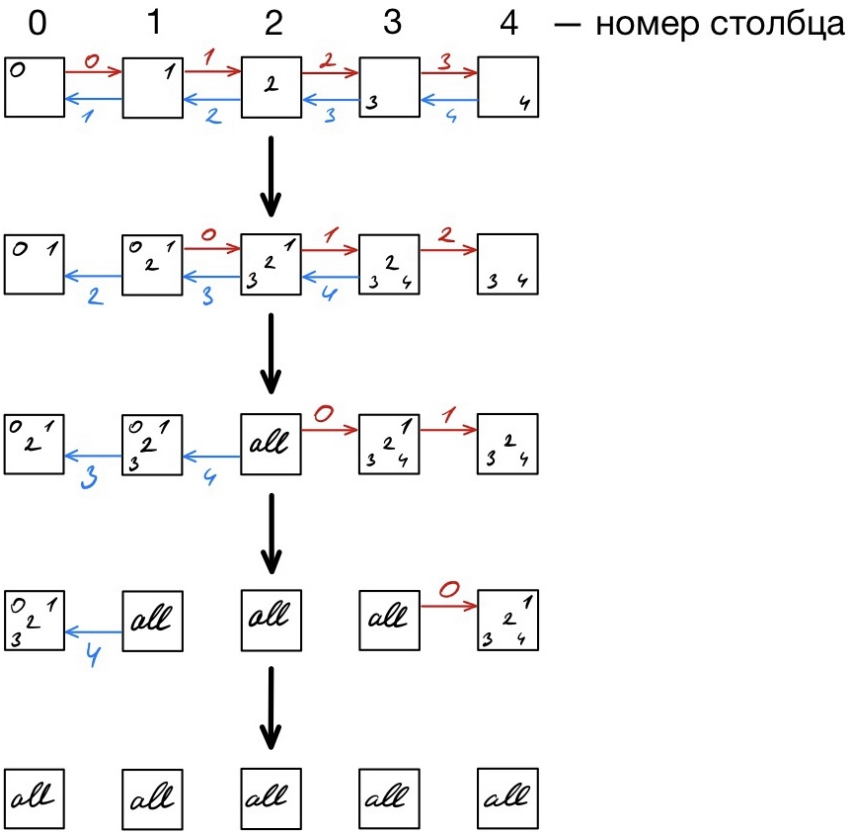

Рис. 1: Рассылка буферов вдоль строк

На каждой итерации процессы сохраняли необходимый столбец значений в буфер $neighbors\_buf$, который потом использовался для пересылки вдоль столбцов

На примере матриц 3*3:

a_0	a_1	a_2
a_3	a_4	a_5
a_6	a_7	a_8

b_0	b_1	b_2
b_3	b_4	b_5
b_6	b_7	b_8

c_0	c_1	c_2
c_3	c_4	c_5
c_6	c_7	c_8

В результате пересылок буферов вдоль строк буфер $neighbors\_buf$ в каждом процессе имеет вид:

a_0	b_0	c_0
a_3	b_3	c_3
a_6	b_6	c_6

a_1	b_1	c_1
a_4	b_4	c_4
a_7	b_7	c_7

a_2	b_2	c_2
a_5	b_5	c_5
a_8	b_8	c_8

Далее переход на второй шаг, в котором начиналась пересылка буферов вдоль столбцов. Для этого по аналогии с рассылкой вдоль строк, пересылка буферов происходила внутри цикла *while* с итераторами *up* и *down*, которые обозначали границы использования функций пересылок MPI_Send, MPI_Recv и MPI_Sendrecv_replace. Это позволяет избежать дедлок процессов.

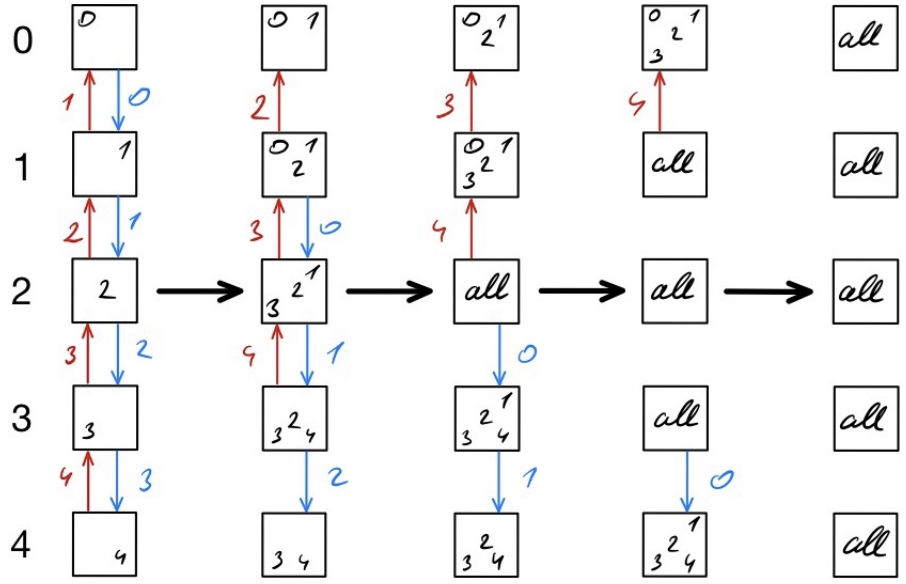

Рис. 2: рассыка буферов вдоль столбцов

Рассмотрим также на примере матриц размера 3*3, только представим первый столбец в горизонтальном виде (в данном случае происходит пересылка буферов между процессами рангов 0, 3, 6 соответственно):

a_0	b_0	c_0
a_3	b_3	c_3
a_6	b_6	c_6

d_0	e_0	f_0
d_3	e_3	f_3
d_6	e_6	f_6

g_0	h_0	i_0
g_3	h_3	i_3
g_6	h_6	i_6

В результате пересылок буфера $neighbors\_buf$ вдоль строк и последующего копирования необходимых строк в буфер my_buf получим результаты для процессов 0, 3 и 6:

a_0	b_0	c_0
d_0	e_0	f_0
g_0	h_0	i_0

a_3	b_3	c_3
d_3	e_3	f_3
g_3	h_3	i_3

a_6	b_6	c_6
d_6	e_6	f_6
g_6	h_6	i_6

Таким образом реализуется алгоритм функции MPI_ALLTOALL, в котором каждый i-й процесс доставит свой j-й элемент процессу с рангом j в позицию буфера с номером i.

Код программы

Вспомогательные функции для копирования значений буферов целиком, копирование определённых строк столбцов.

```
24  void copy_buf(int* a, int* b, int s) {
25    for (int i = 0; i < s; i++) {
26      a[i] = b[i];
27    }
28  }
29
30  // Копирование буфера по столбцам (нужно при движени по горизонтали)
31  void copy_recv_buf_to_next_buf(int* a /* Что копируем */,
32                                 int* b /* Куда копируем */,
33                                 int num_a,
34                                 int num_b) {
35    for (int i = 0; i < SIZE; i++) {
36      b[i * SIZE + num_a] = a[i * SIZE + num_b];
37    }
38  }
39
40  // Копирование буфера по строкам (нужно при движении по вертикали)
41  void copy_recv_buf_to_result_buf(int* a /* Что копируем */,
42                                   int* b /* Куда копируем */,
43                                   int num_a,
44                                   int num_b) {
45    for (int i = 0; i < SIZE; i++) {
46      b[num_a * SIZE + i] = a[num_b * SIZE + i];
47    }
48  }
```

Рис. 3: Вспомогательные функции

Далее, инициализация процессов MPI и последующее выделение памяти для всех буферов.

```
52  MPI_Init(&argc, &argv);
53  int number_of_processes, rank;
54
55  MPI_Status status;
56
57  MPI_Comm_size(MPI_COMM_WORLD, &number_of_processes);
58  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
59
60  int num_row = rank % SIZE;
61  int num_col = rank / SIZE;
62  int total_s = SIZE * SIZE;
63  int *my_buf, *buf_to_right, *buf_to_left;
64
65  // Буферы:
66  // my_buf - изначальный буфер в процессе. В нём же и будет результат
67  // buf_to_* - буфер, который передаётся по направлению (right, left, up, down)
68  // neighbors_buf - буфер, который хранит в себе промежуточные значения после
69  // пересылок по горизонтали
70  my_buf = calloc(total_s + 1, sizeof(*my_buf));
71  buf_to_right = calloc(total_s + 1, sizeof(*buf_to_right));
72  buf_to_left = calloc(total_s + 1, sizeof(*buf_to_left));
73
74  int* neighbors_buf;
75  neighbors_buf = calloc(total_s + 1, sizeof(*neighbors_buf));
76
77  // Задание начального буфера
78  for (int i = 0; i < total_s; i++) {
79      my_buf[i] = (rank + 1) * i;
80  }
```

Рис. 4: Выделение памяти для матриц и заполнение

Алгоритм пересылки буферов вдоль строк.

```
106     // Последний элемент буфера хранит номер его столбца
107     my_buf[total_s] = num_row;
108     neighbors_buf[total_s] = num_col;
109
110     copy_buf(buf_to_right, my_buf, total_s + 1);
111     copy_buf(buf_to_left, my_buf, total_s + 1);
112
113     // Необходимое копирование, чтобы процесс не потерял свой же столбец элементов
114     // в neighbors_buf
115     copy_recv_buf_to_next_buf(buf_to_right, neighbors_buf, num_row, num_row);
116
117     int left = 0, right = SIZE - 1;
118     const int c_left = 0, c_right = SIZE - 1;
119
120     // Рассылка буферов вдоль строк
121     while (left < c_right && right > c_left) {
122       // Движение вправо
123       if (num_row == left) {
124         MPI_Send(buf_to_right, total_s + 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
125       } else if (num_row == c_right) {
126         MPI_Recv(buf_to_right, total_s + 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
127                  &status);
128       } else if (num_row > left && num_row < c_right) {
129         MPI_Sendrecv_replace(buf_to_right, total_s + 1, MPI_INT, rank + 1, 0,
130                              rank - 1, 0, MPI_COMM_WORLD, &status);
131       }
132
133       // Движение влево
134       if (num_row == right) {
135         MPI_Send(buf_to_left, total_s + 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD);
136       } else if (num_row == c_left) {
137         MPI_Recv(buf_to_left, total_s + 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD,
138                  &status);
139       } else if (num_row > c_left && num_row < right) {
140         MPI_Sendrecv_replace(buf_to_left, total_s + 1, MPI_INT, rank - 1, 0,
141                              rank + 1, 0, MPI_COMM_WORLD, &status);
142       }
143
144       int tmp_num_right = buf_to_right[total_s];
145       int tmp_num_left = buf_to_left[total_s];
146
147       copy_recv_buf_to_next_buf(buf_to_right, neighbors_buf, tmp_num_right,
148                                 num_row);
149       copy_recv_buf_to_next_buf(buf_to_left, neighbors_buf, tmp_num_left,
150                                 num_row);
151
152       left++;
153       right--;
154     }
```

Рис. 5: Движение по гориизонтали

После сразу алгоритм пересылки буферов вдоль столбцов.

```
156     int up = 0, down = SIZE - 1;
157     int c_up = 0, c_down = SIZE - 1;
158     int *buf_to_down, *buf_to_up;
159     buf_to_down = calloc(total_s + 1, sizeof(*buf_to_down));
160     buf_to_up = calloc(total_s + 1, sizeof(*buf_to_up));
161
162     copy_buf(buf_to_up, neighbors_buf, total_s + 1);
163     copy_buf(buf_to_down, neighbors_buf, total_s + 1);
164
165     // Необходимое копирование, чтобы процесс не потерял свою же строку элементов
166     // в my_result
167     copy_recv_buf_to_result_buf(buf_to_down, my_buf, num_col, num_col);
168
169     // Рассылка буферов вдоль столбцов
170     while (up < c_down && down > c_up) {
171       // Движение вниз
172       if (num_col == up) {
173         MPI_Send(buf_to_down, total_s + 1, MPI_INT, rank + SIZE, 0,
174                  MPI_COMM_WORLD);
175       } else if (num_col == c_down) {
176         MPI_Recv(buf_to_down, total_s + 1, MPI_INT, rank - SIZE, 0,
177                  MPI_COMM_WORLD, &status);
178       } else if (num_col > up && num_col < c_down) {
179         MPI_Sendrecv_replace(buf_to_down, total_s + 1, MPI_INT, rank + SIZE, 0,
180                              rank - SIZE, 0, MPI_COMM_WORLD, &status);
181       }
182
183       // Движение вверх
184       if (num_col == down) {
185         MPI_Send(buf_to_up, total_s + 1, MPI_INT, rank - SIZE, 0, MPI_COMM_WORLD);
186       } else if (num_col == c_up) {
187         MPI_Recv(buf_to_up, total_s + 1, MPI_INT, rank + SIZE, 0, MPI_COMM_WORLD,
188                  &status);
189       } else if (num_col > c_up && num_col < down) {
190         MPI_Sendrecv_replace(buf_to_up, total_s + 1, MPI_INT, rank - SIZE, 0,
191                              rank + SIZE, 0, MPI_COMM_WORLD, &status);
192       }
193
194       int tmp_num_down = buf_to_down[total_s];
195       int tmp_num_up = buf_to_up[total_s];
196
197       copy_recv_buf_to_result_buf(buf_to_down, my_buf, tmp_num_down, num_col);
198       copy_recv_buf_to_result_buf(buf_to_up, my_buf, tmp_num_up, num_col);
199
200       up++;
201       down--;
202     }
```

Рис. 6: Движение по вертикали

Вывод получившегося буфера для каждого процесса, освобождение всей выделенной памяти и завершение работы.

```
204   // Результат
205   MPI_Barrier(MPI_COMM_WORLD);
206   if (rank == 0) {
207     printf("////////////////////////////////////\n");
208     printf("////////     RESULTS:     ////////\n");
209     printf("////////////////////////////////////\n");
210   }
211   MPI_Barrier(MPI_COMM_WORLD);
212
213   for (int i = 0; i < total_s; i++) {
214     if (rank == i) {
215       printf("rank = %d:\n", rank);
216       for (int j = 0; j < total_s; j++) {
217         printf("%d ", my_buf[j]);
218         if ((j + 1) % SIZE == 0) {
219           printf("\n");
220         }
221       }
222       printf("\n");
223       fflush(stdout);
224     }
225     MPI_Barrier(MPI_COMM_WORLD);
226   }
227
228   free(neighbors_buf);
229
230   free(buf_to_right);
231   free(buf_to_left);
232   free(buf_to_up);
233   free(buf_to_down);
234   free(my_buf);
```

Рис. 7: Результат программы и завершение работы

Оценка времени работы программы

Время старта равно 100, время пересылки одного байта равно 1 ($Ts = 100,\ Tb = 1$). Время работы программы можно оценить по формуле:

$$time = \sum_{i=1}^{cnt}(Ts + n_i \cdot Tb),$$

где cnt это количество шагов, n_i - количество пересылаемых байт за одну пересылку на каждом шаге алгоритма.

Если учитывать, что таких шагов у нас всего 8 (4 при пересылке по горизонали и 4 по вертикали), и на каждом шаге процессы пересылают 104 байт данных (одномерный буфер размера 25*4 байт и еще дополнительный элемент на 25-й позиции в буфере, обозначающий номер процесса, которому этот буфер принадлежал), то можно высчитать:

$$time = \sum_{i=1}^{8}(100 + n_i \cdot 1) = 1632,$$

так как $n_i = 104 \ \forall \ i \in \{1, ..., 8\}$.

Вывод программы

```
rank = 0:
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

rank = 1:
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25

rank = 2:
2 4 6 8 10
12 14 16 18 20
22 24 26 28 30
32 34 36 38 40
42 44 46 48 50

rank = 3:
3 6 9 12 15
18 21 24 27 30
33 36 39 42 45
48 51 54 57 60
63 66 69 72 75

rank = 4:
4 8 12 16 20
24 28 32 36 40
44 48 52 56 60
64 68 72 76 80
84 88 92 96 100

rank = 5:
5 10 15 20 25
30 35 40 45 50
55 60 65 70 75
80 85 90 95 100
105 110 115 120 125

rank = 6:
6 12 18 24 30
36 42 48 54 60
66 72 78 84 90
96 102 108 114 120
126 132 138 144 150

rank = 7:
7 14 21 28 35
42 49 56 63 70
77 84 91 98 105
112 119 126 133 140
147 154 161 168 175

rank = 8:
8 16 24 32 40
48 56 64 72 80
88 96 104 112 120
128 136 144 152 160
168 176 184 192 200

rank = 9:
9 18 27 36 45
54 63 72 81 90
99 108 117 126 135
144 153 162 171 180
189 198 207 216 225

rank = 10:
10 20 30 40 50
60 70 80 90 100
110 120 130 140 150
160 170 180 190 200
210 220 230 240 250

rank = 11:
11 22 33 44 55
66 77 88 99 110
121 132 143 154 165
176 187 198 209 220
231 242 253 264 275

rank = 12:
12 24 36 48 60
72 84 96 108 120
132 144 156 168 180
192 204 216 228 240
252 264 276 288 300

rank = 13:
13 26 39 52 65
78 91 104 117 130
143 156 169 182 195
208 221 234 247 260
273 286 299 312 325

rank = 14:
14 28 42 56 70
84 98 112 126 140
154 168 182 196 210
224 238 252 266 280
294 308 322 336 350

rank = 15:
15 30 45 60 75
90 105 120 135 150
165 180 195 210 225
240 255 270 285 300
315 330 345 360 375

rank = 16:
16 32 48 64 80
96 112 128 144 160
176 192 208 224 240
256 272 288 304 320
336 352 368 384 400

rank = 17:
17 34 51 68 85
102 119 136 153 170
187 204 221 238 255
272 289 306 323 340
357 374 391 408 425

rank = 18:
18 36 54 72 90
108 126 144 162 180
198 216 234 252 270
288 306 324 342 360
378 396 414 432 450

rank = 19:
19 38 57 76 95
114 133 152 171 190
209 228 247 266 285
304 323 342 361 380
399 418 437 456 475

rank = 20:
20 40 60 80 100
120 140 160 180 200
220 240 260 280 300
320 340 360 380 400
420 440 460 480 500

rank = 21:
21 42 63 84 105
126 147 168 189 210
231 252 273 294 315
336 357 378 399 420
441 462 483 504 525

rank = 22:
22 44 66 88 110
132 154 176 198 220
242 264 286 308 330
352 374 396 418 440
462 484 506 528 550

rank = 23:
23 46 69 92 115
138 161 184 207 230
253 276 299 322 345
368 391 414 437 460
483 506 529 552 575

rank = 24:
24 48 72 96 120
144 168 192 216 240
264 288 312 336 360
384 408 432 456 480
504 528 552 576 600
```

Рис. 8: Вывод программы

Задание №2

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: а) продолжить работу программы только на “исправных” процессах; б) вместо процессов, вышедших из строя, создать новые MPI-процессы, которые необходимо использовать для продолжения расчетов; в) при запуске программы на счет сразу запустить некоторое дополнительное количество MPI-процессов, которые использовать в случае сбоя.

Подход к решению

Была выбрана реализация пункта В), связанная с запуском дополнительных MPI процессов.

При запуске программы сразу запускается дополнительный MPI-процесс, который используется в случае сбоя. К программе были добавлены функции, сохраняющие промежуточное состояние блоков матриц в файлы, соответствующие исполняемому процессу, и функции выкачки этих матриц и последнего рабочего состояния. Был добавлен обработчик, который в случае сбоя удаляет из коммуникатора умерший процесс, и вместо него добавляет один из дополнительных. Затем восстанавливаются данные, и основной ход программы продолжает работу с того же места.

Код программы

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

int number_of_processes, rank;
int kill_rank = 3;
MPI_Comm main_comm = MPI_COMM_WORLD;
int error_occured = 0;
```

Обработчик процессов

```
static void err_handler(MPI_Comm* pcomm, int* perr, ...) {
  error_occured = 1;
```

```
  int err = *perr;
  char errstr[MPI_MAX_ERROR_STRING];
  int size, failed_len, len;

  MPI_Group failed_group;
  MPI_Comm_size(main_comm, &size);

  MPIX_Comm_failure_ack(main_comm);
  MPIX_Comm_failure_get_acked(main_comm, &failed_group);

  MPI_Group_size(failed_group, &failed_len);
  MPI_Error_string(err, errstr, &len);
  printf("\nRank %d / %d: Error %s. Dead %d\n", rank, size, errstr, nf);

  // создаем новый коммуникатор без вышедшего из строя процесса
  MPIX_Comm_shrink(main_comm, &main_comm);
  MPI_Comm_rank(main_comm, &rank);

}
```

Функции загрузки и сохранения данных

```
static void data_save() {
  if (myrank == 0) {
    FILE* f = fopen("elimination.bin", "wb");
    fwrite(&A[0], sizeof(double), (N - 1) * N, f);
    fclose(f);

    if (reverse_sub) {
      FILE* f = fopen("reverse_sub.bin", "wb");
      fwrite(&X[0], sizeof(double), N, f);
      fclose(f);
    }
  }
  MPI_Barrier(main_comm);
}

static void data_load() {
  FILE* f = fopen("elimination.bin", "rb");

  fread(&A[0], sizeof(double), (N - 1) * N, f);
  fclose(f);
  printf("Proc %d\n", myrank);

  if (reverse_sub) {
    FILE* f = fopen("reverse_sub.bin", "wb");
```

```
    fwrite(&X[0], sizeof(double), N, f);
    fclose(f);
  }

  MPI_Barrier(main_comm);
}
```

Основной код программы

```
void print_m(int** m, int size) {
  char output_format[] = "%*d ";
  int max_elem = m[size - 1][size - 1];
  int x = 1, st = 0;
  while (x < max_elem) {
    x *= 10;
    st++;
  }
  output_format[1] = st + '0';
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      printf(output_format, m[i][j]);
    }
    printf("\n");
  }
  printf("\n");
}

void get_block(int* tmp, int** m_a, int block_i, int block_j, int block_size) {
  for (int i = 0; i < block_size; i++) {
    for (int j = 0; j < block_size; j++) {
      tmp[i * block_size + j] =
          m_a[block_i * block_size + i][block_j * block_size + j];
    }
  }
}

void buf_to_matr(int* tmp, int** matrix, int block_size) {
  for (int i = 0; i < block_size; i++) {
    for (int j = 0; j < block_size; j++) {
      matrix[i][j] = tmp[i * block_size + j];
    }
  }
}

void matr_to_buf(int* tmp, int** matrix, int block_size) {
  for (int i = 0; i < block_size; i++) {
```

```
    for (int j = 0; j < block_size; j++) {
      tmp[i * block_size + j] = matrix[i][j];
    }
  }
}

void block_to_matrix(int** block,
                     int** matrix,
                     int block_i,
                     int block_j,
                     int block_size) {
  for (int i = 0; i < block_size; i++) {
    for (int j = 0; j < block_size; j++) {
      matrix[block_i * block_size + i][block_j * block_size + j] = block[i][j];
    }
  }
}

void matrix_mul(int** m_a, int** m_b, int** m_c, int size) {
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      for (int k = 0; k < size; k++) {
        m_c[i][j] += m_a[i][k] * m_b[k][j];
      }
    }
  }
}

int main(int argc, char** argv) {
  int matrix_size = atoi(argv[1]);

  MPI_Init(&argc, &argv);

  int number_of_processes, rank;

  MPI_Comm_size(MPI_COMM_WORLD, &number_of_processes);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // обработчик ошибок
  MPI_Errhandler errh;
  MPI_Comm_create_errhandler(err_handler, &errh);
  MPI_Comm_set_errhandler(MPI_COMM_WORLD, errh);
  MPI_Barrier(MPI_COMM_WORLD);

  create_filename(rank, filename);
  strcat(filename, ".txt");
```

```
if (matrix_size * matrix_size % (number_of_processes - 1) != 0) {
  printf("Incorrect Number of processes\n");
  return 0;
}
int block_size = matrix_size * matrix_size / (number_of_processes - 1);
block_size = sqrt(block_size);
int block_cnt = matrix_size / block_size;

if (rank == 0) {
  // Выделение памяти для исходных матриц A и B и результирующей матрицы C
  int **matrix_a, **matrix_b, **matrix_c;

  matrix_a = calloc(matrix_size, sizeof(*matrix_a));
  matrix_b = calloc(matrix_size, sizeof(*matrix_b));
  matrix_c = calloc(matrix_size, sizeof(*matrix_c));

  for (int i = 0; i < matrix_size; i++) {
    matrix_a[i] = calloc(matrix_size, sizeof(*matrix_a[i]));
    matrix_b[i] = calloc(matrix_size, sizeof(*matrix_b[i]));
    matrix_c[i] = calloc(matrix_size, sizeof(*matrix_c[i]));
  }

  // Заполнения матриц A и B
  for (int i = 0; i < matrix_size; i++) {
    for (int j = 0; j < matrix_size; j++) {
      matrix_a[i][j] = i * matrix_size + j;
      matrix_b[i][j] = matrix_a[i][j];
    }
  }

  double start_time = MPI_Wtime();

  int* tmp = calloc(block_size * block_size, sizeof(*tmp));

  // Рассылка блоков Aij и Bij в соответствующие процессы
  for (int p = 1; p < number_of_processes; p++) {
    int i = (p - 1) / block_cnt, j = (p - 1) % block_cnt;

    get_block(tmp, matrix_a, i, j, block_size);
    MPI_Send(tmp, block_size * block_size, MPI_INT, p, 0, MPI_COMM_WORLD);

    get_block(tmp, matrix_b, i, j, block_size);
    MPI_Send(tmp, block_size * block_size, MPI_INT, p, 0, MPI_COMM_WORLD);
  }
  MPI_Barrier(MPI_COMM_WORLD);
```

```
  int** get_block;
  get_block = calloc(block_size, sizeof(*get_block));
  for (int k = 0; k < block_size; k++) {
    get_block[k] = calloc(block_size, sizeof(**get_block));
  }

  for (int p = 1; p < number_of_processes; p++) {
    int i = (p - 1) / block_cnt, j = (p - 1) % block_cnt;

    // Получение получившихся блоков Cij из каждого процесса
    MPI_Recv(tmp, block_size * block_size, MPI_INT, p, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    buf_to_matr(tmp, get_block, block_size);
    // Вставляем этот блок в матрицу C
    block_to_matrix(get_block, matrix_c, i, j, block_size);
  }
  // Замер времени
  double end_time = MPI_Wtime();
  printf("Time taken to execute is\n%lf\n", end_time - start_time);

  // print_m(matrix_c, matrix_size);

  // Освобождение памяти
  free(tmp);
  for (int i = 0; i < block_size; i++) {
    free(get_block[i]);
  }
  free(get_block);

  for (int i = 0; i < matrix_size; i++) {
    free(matrix_a[i]);
    free(matrix_b[i]);
    free(matrix_c[i]);
  }
  free(matrix_a);
  free(matrix_b);
  free(matrix_c);
} else if (rank != 0) {
  int i = (rank - 1) / block_cnt, j = (rank - 1) % block_cnt;
  int* tmp;    // Для разных переносимых буферов
  int* buf_a;  // Хранит свой блок Aij в виде буфера
  tmp = calloc(block_size * block_size, sizeof(*tmp));
  buf_a = calloc(block_size * block_size, sizeof(*tmp));

  int **get_a, **get_b, **get_c;  // Для хранения блоков Aij, Bij и Cij
```

```
get_a = calloc(block_size, sizeof(*get_a));
get_b = calloc(block_size, sizeof(*get_b));
get_c = calloc(block_size, sizeof(*get_c));

for (int k = 0; k < block_size; k++) {
  get_a[k] = calloc(block_size, sizeof(**get_a));
  get_b[k] = calloc(block_size, sizeof(**get_b));
  get_c[k] = calloc(block_size, sizeof(**get_c));
}

// Получение от главного процесса блоков Aij и Bij
MPI_Recv(buf_a, block_size * block_size, MPI_INT, 0, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
buf_to_matr(buf_a, get_a, block_size);

MPI_Recv(tmp, block_size * block_size, MPI_INT, 0, 0, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
buf_to_matr(tmp, get_b, block_size);
MPI_Barrier(MPI_COMM_WORLD);

int m = 0;  // Цикл по m
for (m = 0; m < block_cnt; m++) {
  int col_num_sended_A = (i + m) % block_cnt;
  // Рассылка блока Aij всем процессам в этой строке
  if (block_cnt > 1) {
    if (j == col_num_sended_A) {
      for (int k = 0; k < block_cnt; k++) {
        if (k != j) {
          MPI_Send(buf_a, block_size * block_size, MPI_INT,
                   i * block_cnt + k + 1, 0, MPI_COMM_WORLD);
        }
      }
    } else {
      MPI_Recv(tmp, block_size * block_size, MPI_INT,
               i * block_cnt + col_num_sended_A + 1, 0, MPI_COMM_WORLD,
               MPI_STATUS_IGNORE);
      buf_to_matr(tmp, get_a, block_size);
    }
  }

  // Умножение блоков Aij и Bij, запись результата в Cij
  matrix_mul(get_a, get_b, get_c, block_size);

  if (block_cnt > 1) {
    int send_num = rank - block_cnt;
    int recv_num = rank + block_cnt;
```

```
    if (recv_num >= number_of_processes) {
      recv_num -= (number_of_processes - 1);
    }
    if (send_num <= 0) {
      send_num += number_of_processes - 1;
    }

    matr_to_buf(tmp, get_b, block_size);

    // Циклическая пересылка блока Bij в направление убывания строк.
    // Условия if и else необходимы для избежания Deadlock
    if (i % 2) {
      MPI_Send(tmp, block_size * block_size, MPI_INT, send_num, 0,
               MPI_COMM_WORLD);
      MPI_Recv(tmp, block_size * block_size, MPI_INT, recv_num, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);

      buf_to_matr(tmp, get_b, block_size);
    } else {
      int* tmp_b = calloc(block_size * block_size, sizeof(*tmp));
      matr_to_buf(tmp_b, get_b, block_size);
      MPI_Recv(tmp_b, block_size * block_size, MPI_INT, recv_num, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Send(tmp, block_size * block_size, MPI_INT, send_num, 0,
               MPI_COMM_WORLD);
      buf_to_matr(tmp_b, get_b, block_size);
      free(tmp_b);
    }
  }

  buf_to_matr(buf_a, get_a, block_size);
}

matr_to_buf(tmp, get_c, block_size);

// Сохранение данных и убийство процесса
if (rank = kill_rank) {
  data_save();
  raise(SIGKILL);
}
MPI_Barrier(MPI_COMM_WORLD);

// При существовании ошибки нужный ранг забирает данные и высылает их
if (error_occured && rank == number_of_processes - 1) {
  error_occured = 0;
```

```
      data_load();
    }
    // Пересылка получившегося блока Cij в главный процесс с номером 0
    MPI_Send(tmp, block_size * block_size, MPI_INT, 0, 0, MPI_COMM_WORLD);

    // Освобождение всей памяти в процессе
    for (int k = 0; k < block_size; k++) {
      free(get_a[k]);
      free(get_b[k]);
      free(get_c[k]);
    }
    free(get_a);
    free(get_b);
    free(get_c);
    free(tmp);
    free(buf_a);
  }

  MPI_Finalize();
  return 0;
}
```