Отчёт по практическому заданию «Распределенные системы»

Утешева Екатерина Евгеньевна
421

2025

Содержание

1 Введение

В рамках данного задания были реализованы две MPI-программы:

1. Моделирование передачи длинного сообщения в транспьютерной матрице 8×8 с использованием конвейерной передачи по двум параллельным маршрутам.

2. Модификация MPI-программы для задачи релаксации с добавлением устойчивости к сбоям через механизм контрольных точек с использованием MPI-IO и ULFM.

2 Задание №1: Передача сообщения в транспьютерной матрице

2.1 Постановка задачи

Требуется смоделировать передачу длинного сообщения (L - длина) из узла $(0, 0)$ в узел $(7, 7)$ в транспьютерной матрице 8×8, где каждый узел представляет собой отдельный MPI-процесс. Параметры модели:

$$T_s = 100, \quad T_b = 1.$$

Для эффективного использования пропускной способности каналов длинное сообщение разбить на части, которые передаются по нескольким маршрутам одновременно.

2.2 Описание алгоритма

1. Инициализация сетки 8×8 (64 процесса)
2. Разбиение сообщения на K (CHUNK_COUNT) частей
3. Параллельная отправка по двум независимым "Г-образным"маршрутам:
   - Чётные части горизонтально, затем вертикально

     $$(0, 0) \to (1, 0) \to \cdots \to (7, 0)$$

     $$(7, 0) \to (7, 1) \to \cdots \to (7, 7)$$

   - Нечётные части вертикально, затем горизонтально

     $$(0, 0) \to (0, 1) \to \cdots \to (0, 7)$$

     $$(0, 7) \to (1, 7) \to \cdots \to (7, 7)$$
4. Конвейерная передача через промежуточные узлы
5. Сборка сообщения в узле назначения

Для передачи данных между процессами используется неблокирующая функция MPI_Isend, для получения MPI_Irecv. Для ожидания завершения выполнения функций используется MPI_Waitall.

Например, на узле получателе:

```
MPI_Request* requests = malloc(CHUNK_COUNT * sizeof(MPI_Request));
for (int chunk = 0; chunk < CHUNK_COUNT; chunk++) {
    // Приём сообщений
    MPI_Irecv(message + offset, chunk_size, MPI_CHAR,
              prev_rank, chunk, MPI_COMM_WORLD, &requests[chunk]);
}
// Ждем все запросы
MPI_Waitall(CHUNK_COUNT, requests, MPI_STATUSES_IGNORE);
```

2.3 Временная оценка

Рассмотрим время передачи сообщения длиной L по маршруту длиной d хопов с разбиением на K частей при использовании двух параллельных маршрутов.

Время, необходимое для пересылки одного куска сообщения от (0,0) до (7,7):

$$T_1 = (L \cdot T_b)/K \cdot d$$

Так как сообщение очень длинное, то можно пренебречь длиной маршрута, временем старта и временем разгона конвейера. Получаем:

$$T_1 \approx \frac{L \cdot T_b}{K}$$

Поскольку передача на каждом из хопов происходит последовательно, общее время доставки сообщения от источника к получателю равно времени доставки последнего куска до конечной точки:

$$T = (K-1) \cdot (T_b \cdot L/K) + (T_b \cdot L/K) = T_b \cdot L = L$$

2.4 Инструкция по запуску

```
mpicc -o matrix_v2 matrix_v2.c
mpirun -np 64 ./matrix_v2
```

Программа требует ровно 64 процесса (сетка 8×8).

Исходный код доступен в Приложении 1.

3 Задание №2: Устойчивая к сбоям программа с контрольными точками

3.1 Постановка задачи

Требуется доработать MPI-программу для задачи релаксации двумерной матрицы, добавив механизм контрольных точек через MPI-IO для возможности восстановления после сбоя. Реализован сценарий (б):

> Вместо процессов, вышедших из строя, динамически создаются новые MPI-процессы через механизм MPI_Comm_spawn, которые загружают контрольную точку и продолжают вычисления.

3.2 Описание алгоритма

1. Инициализация с распределением данных

   Матрица $N \times N$ (где $N = 4096 + 2$) разбивается по строкам между процессами, после чего каждый процесс выделяет локальный массив. Таким образом, ни один процесс не хранит массив целиком.

2. Попытка загрузки checkpoint

3. Основной цикл релаксации:

   - Неблокирующий обмен граничными строками (MPI_Isend, MPI_Irecv)

- Вычисление новых значений
- Подсчёт локального изменения и вычисление глобального изменения (MPI_Allreduce).
- Проверка сходимости
- Периодическое сохранение checkpoint (с использованием MPI_File)

4. При сбое обработка через ULFM (void error_handler)
   - Аннулирование коммуникатора (MPIX_Comm_revoke)
   - Создание нового коммуникатора без сбойных процессов (MPIX_Comm_shrink)
   - Динамическое создание новых процессов (MPI_Comm_spawn) и объединение коммуникаторов (MPI_Intercomm_merge)
   - Пересчёт распределения и выделение памяти
   - Загрузка checkpoint
     Каждый процесс читает свою часть данных из файла по формуле:

$$\text{offset} = 3 \cdot \text{sizeof(int)} + \sum_{r=0}^{\text{rank}-1} h_r \cdot N \cdot \text{sizeof(double)}$$

     где h_r — высота данных процесса r.
   - Возврат к основному циклу
5. Верификация результата

Новые процессы выполняют следующие действия:

1. Получают parent communicator через `MPI_Comm_get_parent()`
2. Объединяются с родительским коммуникатором через `MPI_Intercomm_merge`
3. Получают метаданные (пути и номер итерации) через `MPI_Bcast`
4. Вычисляют своё распределение данных
5. Загружают checkpoint
6. Присоединяются к основному циклу релаксации

Физический сбой имитируется с помощью SIGKILL в основном цикле при задании параметров при запуске

3.3 Временная оценка

Рассмотрим временные характеристики программы с учётом контрольных точек и возможных сбоев.

Обозначим t_{iter} время одной итерации без учёта операций ввода-вывода (IO). При параллельном запуске на p процессах время одной итерации можно оценить как:

$$t_{\text{iter}} \approx \frac{N^2}{p \cdot R_{\text{CPU}}}$$

где R_{CPU} — скорость выполнения арифметических операций.

3.3.1 Время вычислений без сбоя

Так как чтение из памяти и запись в память считаются бесконечно быстрыми, то при выполнении программы до сходимости или до максимального числа итераций $it_{\max}$ общее время вычислений можно приблизительно оценить как:

$$T_{\text{total}} = it_{\max} \cdot t_{\text{iter}} = it_{\max} \cdot \frac{N^2}{p \cdot R_{\text{CPU}}}$$

3.3.2 Время восстановления при сбое

Пусть сбой происходит на итерации it_{fail}. Программа откатывается к последнему сохранённому checkpoint на итерации:

$$it_{\text{cp}} = \left\lfloor \frac{it_{\text{fail}}}{c} \right\rfloor \times c$$

Потеря работы составляет:

$$\Delta it = it_{\text{fail}} - it_{\text{cp}}$$

Время восстановления:

$$T_{\text{recovery}} = \underbrace{t_{\text{ULFM}}}_{\text{обработка сбоя}} + \underbrace{t_{\text{spawn}}}_{\text{создание процессов}} + \underbrace{t_{\text{load}}}_{\text{загрузка checkpoint}} \tag{1}$$

$$\approx t_{\text{spawn}} \quad \text{(остальные слагаемые пренебрежимо малы)} \tag{2}$$

Потерянные итерации:

$$\Delta it = it_{\text{fail}} - it_{\text{cp}}$$

Общее время при сбое:

$$T_{\text{with\_1\_error}} = it_{\text{cp}} \cdot t_{\text{iter}}^{(p)} + T_{\text{recovery}} + (it_{\max} - it_{\text{cp}}) \cdot t_{\text{iter}}^{(p)}$$

3.4 Инструкция по запуску

Компиляция (требуется MPI с поддержкой ULFM):

```
mpicc -o mpi_v2 mpi_checkpoint_v2.c
```

Запуск без сбоя:

```
mpirun -np 4 --oversubscribe \
       --with-ft mpi \
       ./mpi_v2
```

Запуск с имитацией сбоя (процесс 1 упадёт на итерации 25):

```
mpirun -np 4 --oversubscribe \
       --mca orte_enable_recovery 1 \
       --with-ft mpi \
       ./mpi_spawn 25 1
```

Исходный код доступен в Приложении 2.

4 Заключение

В ходе работы были успешно реализованы:

- Программа передачи длинного сообщения в транспьютерной матрице с использованием конвейерной передачи по двум параллельным маршрутам. Разбиение сообщения на части и использование неблокирующих операций позволяет эффективно использовать пропускную способность каналов.
- Устойчивая к сбоям MPI-программа с поддержкой контрольных точек через MPI-IO. Использование механизма ULFM позволяет обрабатывать реальные сбои процессов (SIGKILL) и продолжать выполнение программы. При сбое создаются новые процессы и происходит загрузка контрольных точек.

5 Приложение 1. matrix_v2.c

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

#define GRID_SIZE 8
#define MESSAGE_LENGTH 1000000
#define CHUNK_COUNT 50
#define CHUNK_SIZE (MESSAGE_LENGTH / CHUNK_COUNT)

typedef struct {
    int x;
    int y;
} Coord;

typedef enum {
    ROUTE_HORIZONTAL_FIRST = 0,
    ROUTE_VERTICAL_FIRST = 1
} Route;

int coord_to_rank(int x, int y) {
    return y * GRID_SIZE + x;
}

Coord rank_to_coord(int rank) {
    Coord c;
    c.x = rank % GRID_SIZE;
    c.y = rank / GRID_SIZE;
    return c;
}

int manhattan_distance(Coord from, Coord to) {
    return abs(to.x - from.x) + abs(to.y - from. y);
}


int is_on_route(Coord my_coord, Coord source, Coord target, Route route) {
    if (route == ROUTE_HORIZONTAL_FIRST) {
        if (my_coord.y == source.y && my_coord.x > source.x && my_coord.x <=
            ↪  target.x) {
            return 1;
        }
        if (my_coord.x == target.x && my_coord.y > source.y && my_coord.y <
            ↪ target.y) {
            return 2;
        }
    } else {
        if (my_coord.x == source.x && my_coord.y > source.y && my_coord. y
            ↪ <= target.y) {
            return 3;
        }
        if (my_coord.y == target. y && my_coord.x > source.x && my_coord. x
            ↪ < target.x) {
            return 4;
        }
    }
    return 0;
}
```

```
55
56 void get_neighbors(Coord my_coord, Coord source, Coord target, Route route,
57                     int route_position, int *prev_rank, int *next_rank) {
58     if (route == ROUTE_HORIZONTAL_FIRST) {
59         if (route_position == 1) {
60             *prev_rank = coord_to_rank(my_coord.x - 1, my_coord.y);
61             if (my_coord.x < target.x) {
62                 *next_rank = coord_to_rank(my_coord.x + 1, my_coord.y);
63             } else {
64                 *next_rank = coord_to_rank(my_coord.x, my_coord.y + 1);
65             }
66         } else if (route_position == 2) {
67             *prev_rank = coord_to_rank(my_coord.x, my_coord.y - 1);
68             *next_rank = coord_to_rank(my_coord.x, my_coord.y + 1);
69         }
70     } else {
71         if (route_position == 3) {
72             *prev_rank = coord_to_rank(my_coord.x, my_coord.y - 1);
73             if (my_coord.y < target.y) {
74                 *next_rank = coord_to_rank(my_coord.x, my_coord.y + 1);
75             } else {
76                 *next_rank = coord_to_rank(my_coord.x + 1, my_coord.y);
77             }
78         } else if (route_position == 4) {
79             *prev_rank = coord_to_rank(my_coord.x - 1, my_coord.y);
80             *next_rank = coord_to_rank(my_coord.x + 1, my_coord.y);
81         }
82     }
83 }
84
85 int main(int argc, char** argv) {
86     int rank, size;
87     MPI_Status status;
88     double start_time, end_time;
89
90     MPI_Init(&argc, &argv);
91     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
92     MPI_Comm_size(MPI_COMM_WORLD, &size);
93
94     if (size != GRID_SIZE * GRID_SIZE) {
95         if (rank == 0) {
96             printf("Error: Programm need %d processes (grid %dx%d)\n",
97                    GRID_SIZE * GRID_SIZE, GRID_SIZE, GRID_SIZE);
98             printf("Run: mpirun -np %d ./matrix_v2\n",
99                    GRID_SIZE * GRID_SIZE);
100        }
101        MPI_Finalize();
102        return 1;
103    }
104
105    Coord my_coord = rank_to_coord(rank);
106    Coord source = {0, 0};
107    Coord target = {7, 7};
108
109    int source_rank = coord_to_rank(source.x, source.y);
110    int target_rank = coord_to_rank(target. x, target.y);
111    char* message = (char*)malloc(MESSAGE_LENGTH * sizeof(char));
112
113    if (rank == source_rank) {
114        for (int i = 0; i < MESSAGE_LENGTH; i++) {
```

```
115             message[i] = 'A' + (i % 26);
116         }
117 
118         printf("Parameters\n");
119         printf("Source: node (%d, %d) [rank %d]\n", source.x, source.y,
                ↪ source_rank);
120         printf("Destination: node (%d, %d) [rank %d]\n", target.x, target.y,
                ↪  target_rank);
121         printf("Length: %d байт\n", MESSAGE_LENGTH);
122         printf("Number of parts: %d\n", CHUNK_COUNT);
123         printf("Size of one part: %d byte\n", CHUNK_SIZE);
124 
125         start_time = MPI_Wtime();
126 
127         MPI_Request* requests = (MPI_Request*)malloc(CHUNK_COUNT * sizeof(
                ↪ MPI_Request));
128 
129         for (int chunk = 0; chunk < CHUNK_COUNT; chunk++) {
130             int offset = chunk * CHUNK_SIZE;
131             int current_chunk_size = (chunk == CHUNK_COUNT - 1)
132                                      ? (MESSAGE_LENGTH - offset)
133                                      : CHUNK_SIZE;
134 
135             int next_rank;
136             Route route = (chunk % 2 == 0) ? ROUTE_HORIZONTAL_FIRST :
                    ↪ ROUTE_VERTICAL_FIRST;
137 
138             if (route == ROUTE_HORIZONTAL_FIRST) {
139                 next_rank = coord_to_rank(my_coord.x + 1, my_coord.y);
140             } else {
141                 next_rank = coord_to_rank(my_coord.x, my_coord.y + 1);
142             }
143 
144             MPI_Isend(message + offset, current_chunk_size, MPI_CHAR,
145                       next_rank, chunk, MPI_COMM_WORLD, &requests[chunk]);
146         }
147 
148         MPI_Waitall(CHUNK_COUNT, requests, MPI_STATUSES_IGNORE);
149         free(requests);
150 
151         end_time = MPI_Wtime();
152 
153     } else if (rank == target_rank) {
154         start_time = MPI_Wtime();
155 
156         MPI_Request* requests = (MPI_Request*)malloc(CHUNK_COUNT * sizeof(
                ↪ MPI_Request));
157 
158         for (int chunk = 0; chunk < CHUNK_COUNT; chunk++) {
159             int offset = chunk * CHUNK_SIZE;
160             int current_chunk_size = (chunk == CHUNK_COUNT - 1)
161                                      ? (MESSAGE_LENGTH - offset)
162                                      :  CHUNK_SIZE;
163 
164             int prev_rank;
165             Route route = (chunk % 2 == 0) ? ROUTE_HORIZONTAL_FIRST :
                    ↪ ROUTE_VERTICAL_FIRST;
166 
167             if (route == ROUTE_HORIZONTAL_FIRST) {
168                 prev_rank = coord_to_rank(my_coord.x, my_coord.y - 1);
```

```
169             } else {
170                 prev_rank = coord_to_rank(my_coord.x - 1, my_coord.y);
171             }
172 
173             MPI_Irecv(message + offset, current_chunk_size, MPI_CHAR,
174                       prev_rank, chunk, MPI_COMM_WORLD, &requests[chunk]);
175         }
176 
177         MPI_Waitall(CHUNK_COUNT, requests, MPI_STATUSES_IGNORE);
178         free(requests);
179 
180         end_time = MPI_Wtime();
181 
182         int correct = 1;
183         for (int i = 0; i < MESSAGE_LENGTH && correct; i++) {
184             if (message[i] != ('A' + (i % 26))) {
185                 correct = 0;
186             }
187         }
188 
189         printf("\nResults\n");
190         printf("Recieved: %s\n", correct ? "Yes" : "No");
191         printf("Time: %.6f seconds (%.2f mks)\n",
192                end_time - start_time, (end_time - start_time) * 1e6);
193 
194     } else {
195         int route1_pos = is_on_route(my_coord, source, target,
            ↪ ROUTE_HORIZONTAL_FIRST);
196         int route2_pos = is_on_route(my_coord, source, target,
            ↪ ROUTE_VERTICAL_FIRST);
197 
198         if (route1_pos || route2_pos) {
199             MPI_Request* recv_requests = (MPI_Request*)malloc(CHUNK_COUNT *
                ↪ sizeof(MPI_Request));
200             MPI_Request* send_requests = (MPI_Request*)malloc(CHUNK_COUNT *
                ↪ sizeof(MPI_Request));
201             int active_chunks = 0;
202 
203             for (int chunk = 0; chunk < CHUNK_COUNT; chunk++) {
204                 Route route = (chunk % 2 == 0) ? ROUTE_HORIZONTAL_FIRST :
                    ↪ ROUTE_VERTICAL_FIRST;
205                 int route_pos = (route == ROUTE_HORIZONTAL_FIRST) ?
                    ↪ route1_pos : route2_pos;
206 
207                 if (route_pos == 0) {
208                     recv_requests[chunk] = MPI_REQUEST_NULL;
209                     send_requests[chunk] = MPI_REQUEST_NULL;
210                     continue;
211                 }
212 
213                 int offset = chunk * CHUNK_SIZE;
214                 int current_chunk_size = (chunk == CHUNK_COUNT - 1)
215                                        ? (MESSAGE_LENGTH - offset)
216                                        :  CHUNK_SIZE;
217 
218                 int prev_rank, next_rank;
219                 get_neighbors(my_coord, source, target, route, route_pos,
220                               &prev_rank, &next_rank);
221 
222                 MPI_Irecv(message + offset, current_chunk_size, MPI_CHAR,
```

```
223                         prev_rank, chunk, MPI_COMM_WORLD, &recv_requests[
                               ↪ chunk]);
224                 active_chunks++;
225             }
226
227             for (int chunk = 0; chunk < CHUNK_COUNT; chunk++) {
228                 if (recv_requests[chunk] == MPI_REQUEST_NULL) continue;
229
230                 MPI_Wait(&recv_requests[chunk], MPI_STATUS_IGNORE);
231                 Route route = (chunk % 2 == 0) ? ROUTE_HORIZONTAL_FIRST :
                       ↪ ROUTE_VERTICAL_FIRST;
232                 int route_pos = (route == ROUTE_HORIZONTAL_FIRST) ?
                       ↪ route1_pos : route2_pos;
233                 int offset = chunk * CHUNK_SIZE;
234                 int current_chunk_size = (chunk == CHUNK_COUNT - 1)
235                                          ? (MESSAGE_LENGTH - offset)
236                                          :  CHUNK_SIZE;
237                 int prev_rank, next_rank;
238                 get_neighbors(my_coord, source, target, route, route_pos,
239                               &prev_rank, &next_rank);
240                 MPI_Isend(message + offset, current_chunk_size, MPI_CHAR,
241                           next_rank, chunk, MPI_COMM_WORLD, &send_requests[
                               ↪ chunk]);
242             }
243
244             MPI_Waitall(CHUNK_COUNT, send_requests, MPI_STATUSES_IGNORE);
245             free(recv_requests);
246             free(send_requests);
247         }
248     }
249
250     free(message);
251     MPI_Finalize();
252     return 0;
253 }
```

6 Приложение 2. mpi_checkpoint_v2.c

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

#define Max(a, b) ((a) > (b) ? (a) : (b))
#define N (4096 + 2)

double maxeps = 0.1e-7;
int itmax = 100;

double **A = NULL, **B = NULL;
double eps;
int rank_world, size_world;
MPI_Comm main_comm;
MPI_Errhandler errh;

int iteration_to_crash = -1, rank_to_crash = -1, it = 1;
static int has_died = 0, after_recovery = 0;
static jmp_buf recovery_jump;

static char program_path[512];
static char checkpoint_path[512];

int local_start_i = 0, local_end_i = 0, local_height = 0;

void calculate_distribution();
void allocate_local_arrays();
void free_local_arrays();
void init();
void relax();
void verify();
void save_checkpoint(int iteration);
int load_checkpoint(int *iteration);
void error_handler(MPI_Comm *comm, int *error_code, ... );
void spawned_process_main(MPI_Comm parent_comm);

int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    char cwd[256];
    getcwd(cwd, sizeof(cwd));
    snprintf(program_path, sizeof(program_path), "%s/%s", cwd, argv[0]);
    snprintf(checkpoint_path, sizeof(checkpoint_path), "%s/checkpoint.dat",
        ↪ cwd);

    MPI_Comm parent_comm;
    MPI_Comm_get_parent(&parent_comm);

    if (parent_comm == MPI_COMM_NULL) {
        MPI_Comm_rank(MPI_COMM_WORLD, &rank_world);
        MPI_Comm_size(MPI_COMM_WORLD, &size_world);
```

```
58
59          if (argc == 3) {
60              iteration_to_crash = atoi(argv[1]);
61              rank_to_crash = atoi(argv[2]);
62          }
63
64          main_comm = MPI_COMM_WORLD;
65          MPI_Comm_create_errhandler(error_handler, &errh);
66          MPI_Comm_set_errhandler(main_comm, errh);
67
68          if (rank_world == 0) {
69              printf("=== Scenario B:  Dynamic Process Spawning ===\n");
70              printf("Processes: %d\n\n", size_world);
71          }
72
73          setjmp(recovery_jump);
74
75          if (! after_recovery) {
76              calculate_distribution();
77              allocate_local_arrays();
78
79              int iteration_start;
80              if (!load_checkpoint(&iteration_start)) {
81                  init();
82                  iteration_start = 1;
83              }
84              it = iteration_start;
85          }
86
87          while (it <= itmax) {
88              if (rank_world == rank_to_crash && it == iteration_to_crash &&
                    ↪  has_died) {
89                  printf("Process %d failure at iteration %d\n", rank_world,
                        ↪ it);
90                  fflush(stdout);
91                  raise(SIGKILL);
92              }
93
94              relax();
95
96              if (rank_world == 0 && it % 10 == 0) {
97                  printf("Iteration %4d, eps = %.10e\n", it, eps);
98              }
99
100             if (it % 10 == 0) save_checkpoint(it);
101             if (eps < maxeps) break;
102
103             MPI_Barrier(main_comm);
104             it++;
105         }
106
107         verify();
108         free_local_arrays();
109
110     } else {
111         spawned_process_main(parent_comm);
112         MPI_Comm_free(&parent_comm);
113     }
114
115     MPI_Finalize();
```

```
116     return 0;
117 }
118
119 void spawned_process_main(MPI_Comm parent_comm)
120 {
121     MPI_Comm merged_comm;
122     MPI_Intercomm_merge(parent_comm, 1, &merged_comm);
123     main_comm = merged_comm;
124
125     MPI_Comm_rank(main_comm, &rank_world);
126     MPI_Comm_size(main_comm, &size_world);
127     MPI_Comm_create_errhandler(error_handler, &errh);
128     MPI_Comm_set_errhandler(main_comm, errh);
129
130     int path_len;
131     MPI_Bcast(&path_len, 1, MPI_INT, 0, main_comm);
132     MPI_Bcast(checkpoint_path, path_len, MPI_CHAR, 0, main_comm);
133     MPI_Bcast(&path_len, 1, MPI_INT, 0, main_comm);
134     MPI_Bcast(program_path, path_len, MPI_CHAR, 0, main_comm);
135     MPI_Bcast(&it, 1, MPI_INT, 0, main_comm);
136
137     calculate_distribution();
138     allocate_local_arrays();
139
140     int iteration_start;
141     if (! load_checkpoint(&iteration_start)) {
142         init();
143         iteration_start = 1;
144     }
145     it = iteration_start;
146
147     while (it <= itmax) {
148         relax();
149         if (it % 10 == 0) save_checkpoint(it);
150         if (eps < maxeps) break;
151         MPI_Barrier(main_comm);
152         it++;
153     }
154
155     verify();
156     free_local_arrays();
157     MPI_Comm_free(&merged_comm);
158 }
159
160 void calculate_distribution()
161 {
162     MPI_Comm_rank(main_comm, &rank_world);
163     int rank_size;
164     MPI_Comm_size(main_comm, &rank_size);
165     local_start_i = rank_world * N / rank_size;
166     local_end_i = (rank_world + 1) * N / rank_size;
167     if (rank_world == rank_size - 1) local_end_i = N;
168     local_height = local_end_i - local_start_i + 2;
169 }
170
171 void allocate_local_arrays()
172 {
173     A = malloc(local_height * sizeof(double*));
174     B = malloc(local_height * sizeof(double*));
175     for (int i = 0; i < local_height; i++) {
```

```
176         A[i] = calloc(N, sizeof(double));
177         B[i] = calloc(N, sizeof(double));
178     }
179 }
180
181 void free_local_arrays()
182 {
183     if (A) {
184         for (int i = 0; i < local_height; i++) free(A[i]);
185         free(A);
186         A = NULL;
187     }
188     if (B) {
189         for (int i = 0; i < local_height; i++) free(B[i]);
190         free(B);
191         B = NULL;
192     }
193 }
194
195 void init()
196 {
197     for (int local_i = 1; local_i < local_height - 1; local_i++) {
198         int global_i = local_start_i + local_i - 1;
199         for (int j = 0; j < N; j++) {
200             A[local_i][j] = (global_i == 0 || global_i == N-1 || j == 0 || j
                ↪  == N-1)
201                             ? 0.0 : (1.0 + global_i + j);
202         }
203     }
204 }
205
206 void relax()
207 {
208     int rank_size;
209     MPI_Comm_size(main_comm, &rank_size);
210     MPI_Request req[4];
211     int req_cnt = 0;
212
213     if (rank_world < rank_size - 1) {
214         MPI_Isend(A[local_height-2], N, MPI_DOUBLE, rank_world+1, 0,
            ↪ main_comm, &req[req_cnt++]);
215         MPI_Irecv(A[local_height-1], N, MPI_DOUBLE, rank_world+1, 0,
            ↪ main_comm, &req[req_cnt++]);
216     }
217     if (rank_world > 0) {
218         MPI_Isend(A[1], N, MPI_DOUBLE, rank_world-1, 0, main_comm, &req[
            ↪ req_cnt++]);
219         MPI_Irecv(A[0], N, MPI_DOUBLE, rank_world-1, 0, main_comm, &req[
            ↪ req_cnt++]);
220     }
221
222     MPI_Waitall(req_cnt, req, MPI_STATUSES_IGNORE);
223
224     int start = (rank_world == 0) ? 2 : 1;
225     int end = (rank_world == rank_size-1) ? local_height-2 : local_height-1;
226
227     for (int i = start; i < end; i++) {
228         for (int j = 1; j < N-1; j++) {
229             B[i][j] = 0.25 * (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])
                ↪ ;
```

```
230         }
231     }
232
233     double local_eps = 0.0;
234     for (int i = start; i < end; i++) {
235         for (int j = 1; j < N-1; j++) {
236             local_eps = Max(local_eps, fabs(A[i][j] - B[i][j]));
237             A[i][j] = B[i][j];
238         }
239     }
240
241     MPI_Allreduce(&local_eps, &eps, 1, MPI_DOUBLE, MPI_MAX, main_comm);
242 }
243
244 void save_checkpoint(int iteration)
245 {
246     MPI_File fh;
247     if (MPI_File_open(main_comm, checkpoint_path, MPI_MODE_WRONLY |
        ↪ MPI_MODE_CREATE,
248                       MPI_INFO_NULL, &fh) != MPI_SUCCESS) return;
249
250     int rank_size;
251     MPI_Comm_size(main_comm, &rank_size);
252
253     if (rank_world == 0) {
254         int meta[3] = {iteration, rank_size, N};
255         MPI_File_write_at(fh, 0, meta, 3, MPI_INT, MPI_STATUS_IGNORE);
256     }
257     MPI_Barrier(main_comm);
258
259     MPI_Offset offset = 3 * sizeof(int);
260     for (int r = 0; r < rank_world; r++) {
261         int h = ((r+1)*N/rank_size) - (r*N/rank_size);
262         if (r == rank_size-1) h = N - (r*N/rank_size);
263         offset += h * N * sizeof(double);
264     }
265
266     int rows = local_end_i - local_start_i;
267     for (int i = 1; i <= rows; i++) {
268         MPI_File_write_at(fh, offset + (i-1)*N*sizeof(double), A[i], N,
            ↪ MPI_DOUBLE, MPI_STATUS_IGNORE);
269     }
270
271     MPI_File_close(&fh);
272 }
273
274 int load_checkpoint(int *iteration)
275 {
276     if (rank_world == 0) {
277         FILE *f = fopen(checkpoint_path, "r");
278         *iteration = f ? 1 : 0;
279         if (f) fclose(f);
280     }
281     MPI_Bcast(iteration, 1, MPI_INT, 0, main_comm);
282     if (! *iteration) return 0;
283
284     MPI_File fh;
285     if (MPI_File_open(main_comm, checkpoint_path, MPI_MODE_RDONLY,
        ↪ MPI_INFO_NULL, &fh) != MPI_SUCCESS)
286         return 0;
```

```
287
288     int meta[3];
289     if (rank_world == 0) MPI_File_read_at(fh, 0, meta, 3, MPI_INT,
            ↪ MPI_STATUS_IGNORE);
290     MPI_Bcast(meta, 3, MPI_INT, 0, main_comm);
291
292     int rank_size;
293     MPI_Comm_size(main_comm, &rank_size);
294     if (meta[1] != rank_size || meta[2] != N) {
295         MPI_File_close(&fh);
296         return 0;
297     }
298
299     MPI_Offset offset = 3 * sizeof(int);
300     for (int r = 0; r < rank_world; r++) {
301         int h = ((r+1)*N/rank_size) - (r*N/rank_size);
302         if (r == rank_size-1) h = N - (r*N/rank_size);
303         offset += h * N * sizeof(double);
304     }
305
306     int rows = local_end_i - local_start_i;
307     for (int i = 1; i <= rows; i++) {
308         MPI_File_read_at(fh, offset + (i-1)*N*sizeof(double), A[i], N,
                ↪ MPI_DOUBLE, MPI_STATUS_IGNORE);
309     }
310
311     MPI_File_close(&fh);
312
313     // Share
314     MPI_Request req[4];
315     int req_cnt = 0;
316     if (rank_world < rank_size-1) {
317         MPI_Isend(A[local_height-2], N, MPI_DOUBLE, rank_world+1, 0,
                ↪ main_comm, &req[req_cnt++]);
318         MPI_Irecv(A[local_height-1], N, MPI_DOUBLE, rank_world+1, 0,
                ↪ main_comm, &req[req_cnt++]);
319     }
320     if (rank_world > 0) {
321         MPI_Isend(A[1], N, MPI_DOUBLE, rank_world-1, 0, main_comm, &req[
                ↪ req_cnt++]);
322         MPI_Irecv(A[0], N, MPI_DOUBLE, rank_world-1, 0, main_comm, &req[
                ↪ req_cnt++]);
323     }
324     MPI_Waitall(req_cnt, req, MPI_STATUSES_IGNORE);
325
326     *iteration = meta[0] + 1;
327     return 1;
328 }
329
330 void verify()
331 {
332     double local_sum = 0.0;
333     for (int i = 1; i < local_height-1; i++) {
334         int gi = local_start_i + i - 1;
335         for (int j = 0; j < N; j++) {
336             local_sum += A[i][j] * (gi+1) * (j+1) / (N*N);
337         }
338     }
339     double sum;
340     MPI_Reduce(&local_sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, main_comm);
```

```
341     if (rank_world == 0) printf("S = %f\n", sum);
342 }
343
344 void error_handler(MPI_Comm *pcomm, int *error_code, ...)
345 {
346     if (has_died) return;
347     has_died = 1;
348
349     MPIX_Comm_revoke(*pcomm);
350     MPI_Comm shrinked;
351     MPIX_Comm_shrink(*pcomm, &shrinked);
352
353     int old_sz, new_sz;
354     MPI_Comm_size(*pcomm, &old_sz);
355     MPI_Comm_size(shrinked, &new_sz);
356     int failed = old_sz - new_sz;
357
358     if (rank_world == 0) {
359         printf("\n=== FAILURE:  %d failed, spawning replacements ===\n",
            ↪ failed);
360     }
361
362     // Spawn
363     MPI_Info info;
364     MPI_Info_create(&info);
365     char cwd[256];
366     if (getcwd(cwd, sizeof(cwd))) MPI_Info_set(info, "wdir", cwd);
367
368     MPI_Comm intercomm;
369     char *argv[1] = {NULL};
370     int *errs = malloc(failed * sizeof(int));
371
372     int rc = MPI_Comm_spawn(program_path, argv, failed, info, 0, shrinked, &
        ↪ intercomm, errs);
373     MPI_Info_free(&info);
374     free(errs);
375
376     if (rc != MPI_SUCCESS) {
377         printf("Error with spawning\n");
378         exit(1);
379     }
380
381     if (rank_world == 0) printf("=== Spawned %d, merging ===\n", failed);
382
383     sleep(1);
384     MPI_Comm new_comm;
385     MPI_Intercomm_merge(intercomm, 0, &new_comm);
386
387     int len = strlen(checkpoint_path) + 1;
388     MPI_Bcast(&len, 1, MPI_INT, 0, new_comm);
389     MPI_Bcast(checkpoint_path, len, MPI_CHAR, 0, new_comm);
390     len = strlen(program_path) + 1;
391     MPI_Bcast(&len, 1, MPI_INT, 0, new_comm);
392     MPI_Bcast(program_path, len, MPI_CHAR, 0, new_comm);
393     MPI_Bcast(&it, 1, MPI_INT, 0, new_comm);
394
395     free_local_arrays();
396     main_comm = new_comm;
397     MPI_Comm_set_errhandler(main_comm, errh);
398     MPI_Comm_rank(main_comm, &rank_world);
```

```
399     MPI_Comm_size(main_comm, &size_world);
400     calculate_distribution();
401     allocate_local_arrays();
402 
403     int iter;
404     if (!load_checkpoint(&iter)) { init(); iter = 1; }
405     it = iter;
406 
407     if (rank_world == 0) printf("=== RECOVERED:  size=%d, iteration=%d ===\n
            ↪ \n", size_world, it);
408 
409     MPI_Comm_free(&intercomm);
410     MPI_Comm_free(&shrinked);
411     after_recovery = 1;
412     longjmp(recovery_jump, 1);
413 }
```