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

Задание по курсу "Распределённые системы"

Отчёт

Выполнил:
Попов Макар Сергеевич
428 группа

Москва, 2022 год

Содержание

1. Постановка задач

1.1. Задание №1

Разработать программу которая реализует заданный алгоритм. Получить временную оценку работы алгоритма.

В транспьютерной матрице размером 8*8, 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 (Ts=100,Tb=1). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

1.2. Задание №2

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.

2. Результаты

2.1. Задание №1

Поскольку в условии задачи не указано число байт области памяти, подвергаемой редукции, будет считать, что оценка времени работы является функцией от длины этой области памяти.

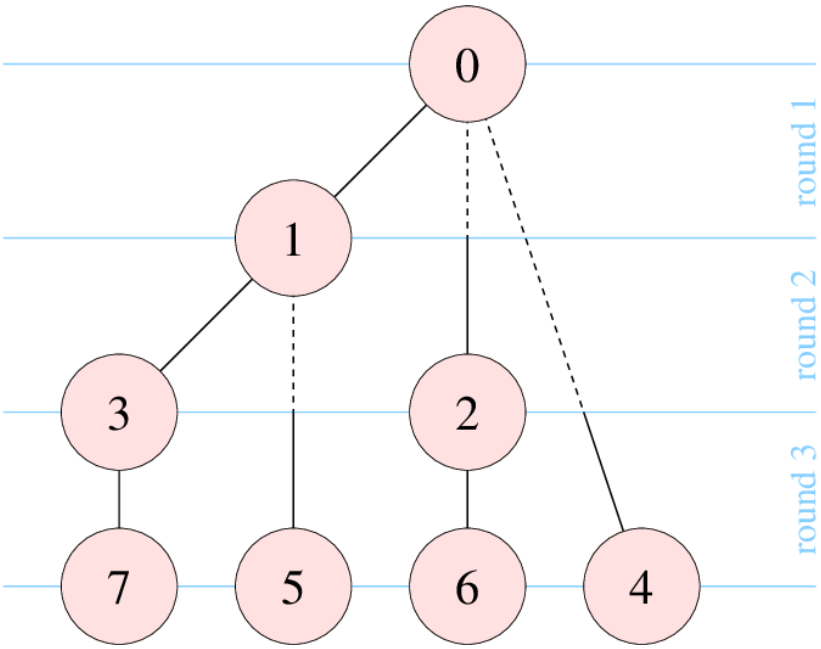
Операцию MPI_MAXLOC разделим на два этапа:

1) редукцию значения с сохранением результата в корневом процессе

2) рассылка значения от корневого процесса остальным

Каждый из этапов операции выполняется посредством блокирующих операций получения и отправки сообщений MPI_Recv и MPI_Send.

Дабы минимизировать затраты времени, обусловленные временем старта передачи, представим транспьютерную матрицу в виде дерева (на примере 8-ми процессов):



Согласно представленной схеме отправки сообщений, на каждом этапе рассылаемая информация отправляется всеми процессами, которые уже ее получили. Для транспьютерной матрицы размера 8*8 потребуется $\log_2(64) = 6$ этапов рассылки значения от корневого процесса остальным.

Редукция значения происходит по той же схеме, но в противоположном направлении, что дает такую же оценку числа этапов пересылки.

Т.о. для области памяти размером n байт и p процессов время выполнения операции MPI_MAXLOC равно:

$$T(n) = T_s * (\lceil \log_2(p) \rceil + T_b * n)$$

В случае $T_s = 100, T_b = 1, p = 64$:

$$T(n) = 600 + 6n$$

.

2.2. Задание №2

MPI-программа, реализованная в рамках курса “Суперкомпьютеры и параллельная обработка данных”, представляет из себя параллельную версию метода Якоби, применяемого над 3-ехмерной матрицей.

Параллельность достигается за счет разделения исходной матрицы на непересекающиеся слои, выделяемые каждому рабочему процессу. В конце каждой итерации требуется синхронизация границ соседних слоев.

В рамках этой работы был реализован сценарий обработки сбоев, согласно которому в начале работы программы сразу запускается некоторое количество дополнительных MPI-процессов, которые используются в случае сбоя.

Принцип работы модифицированной программы состоит в следующем - при запуске всем доступным процессам назначается своя роль: *процесс-мастер*, *рабочий процесс* или *восстановительный процесс*.

2.2.1. Процесс-мастер

Процесс-мастер (который бывает только в единственном экземпляре) отслеживает ход исполнения рабочих процессов.

В случае сбоя какого-либо из них, мастер отправляет запрос восстановительному процессу. Если восстановление прошло успешно, процесс-мастер сообщает соседям упавшего процесса ранг их нового соседа и исполнение возобновляется. Если восстановить процесс не удалось программа аварийно завершает работу.

В случае успешного завершения одного из рабочих процессов, процесс-мастер получает от него сообщение и далее не отслеживает ход исполнения этого процесса. Когда все рабочие процессы завершают работу, процесс-мастер должен принять данные о результате работы каждого из них и сохранить все слои полученной матрицы в своей памяти. Если к этому моменту остался хоть один восстанавливающий процесс, процесс-мастер отправит ему запрос на завершение.

2.2.2. Рабочий процесс

Рабочий процесс выполняет итерационные преобразования над выделенным слоем матрицы и проводит синхронизацию границ своего слоя с соседними процессами в конце каждой итерации. Если в каком-то из рабочих процессов произошел сбой, об этом узнает соседний процесс и передаст запрос на восстановление соседа процессу-мастеру, в ответ на отправленный запрос,

мастер отправит ранг восстановленного соседа. Порядок синхронизации с соседями определяется в момент создания рабочего процесса.

Итерацию алгоритма, выполняемого рабочим процессом можно разделить на три этапа:

- вычисление
- синхронизация с первым соседом (если таковой есть)
- синхронизация со вторым соседом (если таковой есть)

В конце каждого из этапов рабочий процесс создает контрольную точку, начиная с которой исполнение может быть возобновлено в случае сбоя.

После всех итераций алгоритма рабочий процесс должен отправить свой слой матрицы процессу-мастеру и завершить работу.

2.2.3. Восстанавливающий процесс

Наконец, восстанавливающий процесс в начале работы встает на ожидание запроса от процесса-мастера. Если в поступившем запросе указан валидный ранг процесса, то восстанавливающий процесс пытается загрузить контрольную точку упавшего процесса. В случае успеха, восстанавливающий процесс оповещает процесс-мастер и входит в рабочий цикл рабочего процесса.

Если в поступившем от процесса-мастера запросе указан невалидный ранг, это означает, что все рабочие процессы успешно завершили работу, и восстанавливающий процесс поступает также.

3. Исходный код

Весь исходный код программ приведен ниже, но также доступен в репозитории https://github.com/Seg
systems вместе с командами для запуска реализованных программ.

3.1. Задание №1

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <mpi.h>
5  #include <string.h>
6
7  #define N_ROWS 8
8  #define N_COLS 8
9  #define NULL_RANK (-1)
10
11 #define Min(a,b) ((a)<(b)?(a):(b))
12 #define Max(a,b) ((a)>(b)?(a):(b))
13
14 int size = N_ROWS * N_COLS;
15 int rank;
16
17 // https://stackoverflow.com/a/15327567
18 int ceil_log2(unsigned long long x) {
19     static const unsigned long long t[6] = {
20         0xFFFFFFFF00000000ull,
21         0x00000000FFFF0000ull,
22         0x000000000000FF00ull,
23         0x00000000000000F0ull,
24         0x000000000000000Cull,
25         0x0000000000000002ull
26     };
27
28     int y = (((x & (x - 1)) == 0) ? 0 : 1);
29     int j = 32;
30     int i;
31
32     for (i = 0; i < 6; i++) {
33         int k = (((x & t[i]) == 0) ? 0 : j);
34         y += k;
35         x >>= k;
36         j >>= 1;
37     }
38
39     return y;
```

```
40 }
41
42 void init_binomial_heap(int *parent, int *children_num, int **children, int null_parent) {
43     for (int i = 0; i < size; i++) {
44         parent[i] = null_parent;
45     }
46
47     int *queue = calloc(size, sizeof(*queue));
48     int *next_queue = calloc(size, sizeof(*next_queue));
49     int depth = ceil_log2(size);
50     children_num[0] = depth;
51     children[0] = malloc(sizeof(**children) * children_num[0]);
52     queue[0] = children_num[0];
53
54     int next = 1;
55     int reserved = children_num[0];
56     int step = 1;
57
58 #ifdef DEBUG
59     if (rank == 0) { printf("Sequence of messages during broadcast:\n"); }
60 #endif
61     while (reserved) {
62         memcpy(next_queue, queue, size * sizeof(*queue));
63 #ifdef DEBUG
64         if (rank == 0) { printf("step %d: ", step); }
65 #endif
66         for (int i = 0; i < size; i++) {
67             if (queue[i] > 0) {
68                 children[i][children_num[i] - next_queue[i]] = next;
69                 parent[next] = i;
70                 next_queue[i]--;
71 #ifdef DEBUG
72                 if (rank == 0) { printf("%d -> %d ", i, next); }
73 #endif
74                 int child_count = Max(Min(size - next - reserved, next_queue[i]), 0);
75                 if (child_count == 0) {
76                     children[next] = NULL;
77                 } else {
78                     children[next] = malloc(sizeof(**children) * child_count);
79                     next_queue[next] = child_count;
80                     reserved += child_count;
81                 }
82                 children_num[next] = child_count;
83                 next++;
84                 reserved--;
85             }
86         }
```

```
87 #ifdef DEBUG
88         if (rank == 0) { printf("\n"); }
89 #endif
90         memcpy(queue, next_queue, size * sizeof(*queue));
91         step++;
92     }
93 
94     free(next_queue);
95     free(queue);
96 }
97 
98 void reduce_with_max(int *data, int *best_rank, MPI_Comm comm, int parent, int children_num, int *
        children) {
99     if (children_num != 0) {
100         int tmp;
101         int other_rank;
102         for (int i = 0; i < children_num; i++) {
103             MPI_Recv(&tmp, 1, MPI_INT, children[i], 0, comm, MPI_STATUS_IGNORE);
104             MPI_Recv(&other_rank, 1, MPI_INT, children[i], 0, comm, MPI_STATUS_IGNORE);
105             if (tmp > *data) {
106                 *data = tmp;
107                 *best_rank = other_rank;
108             }
109         }
110     }
111     if (parent != NULL_RANK) {
112         MPI_Send(data, 1, MPI_INT, parent, 0, comm);
113         MPI_Send(best_rank, 1, MPI_INT, parent, 0, comm);
114     }
115 }
116 
117 void broadcast_data(int *data, int *best_rank, MPI_Comm comm, int parent, int children_num, int *
        children) {
118     if (parent != NULL_RANK) {
119         MPI_Recv(data, 1, MPI_INT, parent, 0, comm, MPI_STATUS_IGNORE);
120         MPI_Recv(best_rank, 1, MPI_INT, parent, 0, comm, MPI_STATUS_IGNORE);
121     }
122     if (children_num != 0) {
123         for (int i = 0; i < children_num; i++) {
124             MPI_Send(data, 1, MPI_INT, children[i], 0, comm);
125             MPI_Send(best_rank, 1, MPI_INT, children[i], 0, comm);
126         }
127     }
128 }
129 
130 int main(int argc, char *argv[]) {
131     MPI_Init(&argc, &argv);
```

```
132     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
133
134     int *parent_rank = malloc(sizeof(*parent_rank) * size);
135     int *children_num_rank = malloc(sizeof(*children_num_rank) * size);
136     int **children_rank = malloc(sizeof(*children_rank) * size);
137     init_binomial_heap(parent_rank, children_num_rank, children_rank, NULL_RANK);
138
139     MPI_Comm comm;
140     int dims[2] = {N_ROWS, N_COLS};
141     int periods[2] = {0};
142     int coords[2];
143
144     MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
145     MPI_Cart_coords(comm, rank, 2, coords);
146
147     int parent = parent_rank[rank];
148     int children_num = children_num_rank[rank];
149     int *children = children_rank[rank];
150
151     srand(rank);
152     int data = rand() % 1000000;
153     int best_rank = rank;
154
155     if (rank == 0) { printf("Generated data:\n"); }
156     MPI_Barrier(comm);
157
158     for (int i = 0; i < size; i++) {
159         if (i == rank) {
160             printf("rank: %d \tcoords: %d, %d\tdata: %d\n", rank, coords[0], coords[1], data);
161             fflush(stdout);
162         }
163         MPI_Barrier(MPI_COMM_WORLD);
164     }
165
166     reduce_with_max(&data, &best_rank, comm, parent, children_num, children);
167     MPI_Barrier(comm);
168     if (rank == 0) { printf("\nMax data value: %d\nMax data rank: %d\n", data, best_rank); }
169
170     broadcast_data(&data, &best_rank, comm, parent, children_num, children);
171     int best_coords[2];
172     MPI_Cart_coords(comm, best_rank, 2, best_coords);
173     if (rank == 0) { printf("\nData after broadcast:\n"); }
174     MPI_Barrier(comm);
175
176     for (int i = 0; i < size; i++) {
177         if (i == rank) {
178             printf("rank: %d \tbest rank: %d\tbest coords: %d, %d\tdata: %d\n",
```

```
179                     rank, best_rank, best_coords[0], best_coords[1], data);
180             fflush(stdout);
181         }
182         MPI_Barrier(MPI_COMM_WORLD);
183     }
184
185     for (int i = 0; i < size; i++) {
186         if (children_num_rank[i] != 0) {
187             free(children_rank[i]);
188         }
189     }
190     free(children_rank);
191     free(children_num_rank);
192     free(parent_rank);
193
194     MPI_Finalize();
195     return 0;
196 }
```

Листинг 1. transputer_matrix.c

3.2. Задание №2: модифицированная версия

```
1
2  #include <math.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <mpi.h>
6  #include <signal.h>
7  #include <unistd.h>
8
9  #define DEBUG 1
10 #define RECOVERY_PROC_NUM 1
11 #define NULL_RANK (-1)
12 #define NULL_WORKER (-1)
13 #define RECOVERY_IMPOSSIBLE 1
14 #define RECOVERY_FAILED 2
15
16 #define FIRST_SYNC_TAG 1215
17 #define SECOND_SYNC_TAG 1216
18 #define RECOVERY_REQ_TAG 1217
19 #define WORKER_FINISH_TAG 1218
20
21 #define N 34
22 #define MAX_ITERATIONS 100
23
24 #define Max(a,b) ((a)>(b)?(a):(b))
25 #define debug_m_printf if (DEBUG && !rank) printf
26 #define debug_printf if (DEBUG) printf
27
28 #define suicide if (rank == 1 && it_num == 2 && state == PENDING_FIRST_SYNC) { printf("Goodbye...\n"
       ); fflush(stdout); raise(SIGTERM); }
29
30 /*
31  * PROCESS ENTRY FUNCTIONS
32  */
33
34 void master_entry();
35 void recovery_entry();
36 void worker_entry();
37
38 /*
39  * WORKER PROCESS FUNCTIONS
40  */
41
42 void worker_init();
43 void worker_state();
44 void worker_sync(int);
```

```
45 void free_workers_info();
46
47 /*
48  * WORKER RECOVERY FUNCTIONS
49  */
50
51 void save_worker_checkpoint();
52 void load_worker_checkpoint(int);
53 void worker_recovery(int, int);
54
55 /*
56  * MATRIX OPERATION FUNCTIONS
57  */
58
59 void matrix_init();
60 void compute();
61 void relax();
62 void resid();
63 void show_result();
64
65
66 // GENERAL INFO
67 int rank, size;
68 int recovery_proc_num, worker_proc_num;
69
70 // MASTER INFO
71 int *worker_rank, *worker_north_rank, *worker_south_rank, *worker_south_first;
72
73 // WORKER INFO
74 int north_rank, south_rank, south_first;
75 int start_row, last_row, it_num;
76
77 typedef enum {
78     PENDING_COMPUTE,
79     PENDING_FIRST_SYNC,
80     PENDING_SECOND_SYNC
81 } Worker_state;
82
83 Worker_state state;
84
85 double A[N][N][N], B[N][N][N];
86 double eps;
87
88
89 int main(int argc, char **argv) {
90     MPI_Init(&argc, &argv);
91     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
92      MPI_Comm_size(MPI_COMM_WORLD, &size);
93
94      recovery_proc_num = RECOVERY_PROC_NUM;
95      worker_proc_num = size - 1 - recovery_proc_num;
96      debug_m_printf("size: %d, recovery processes: %d, worker processes: %d\n", size,
        recovery_proc_num, worker_proc_num);
97
98      if (worker_proc_num < 1) {
99          debug_m_printf("Not enough worker processes - %d\n", worker_proc_num);
100         MPI_Finalize();
101         return 0;
102     }
103
104     worker_rank = malloc(sizeof(*worker_rank) * worker_proc_num);
105     worker_north_rank = malloc(sizeof(*worker_north_rank) * worker_proc_num);
106     worker_south_rank = malloc(sizeof(*worker_south_rank) * worker_proc_num);
107     worker_south_first = malloc(sizeof(*worker_south_first) * worker_proc_num);
108
109     for (int i = 0; i < worker_proc_num; i++) {
110         int worker_r = i + 1;
111         worker_rank[i] = worker_r;
112         worker_north_rank[i] = worker_r == 1 ? NULL_RANK : worker_r - 1;
113         worker_south_rank[i] = worker_r == worker_proc_num ? NULL_RANK : worker_r + 1;
114         worker_south_first[i] = worker_r & 1;
115         debug_m_printf("worker num: %d, rank: %d, north: %d, south: %d, south_first: %d\n",
116                        i, worker_r, worker_north_rank[i], worker_south_rank[i], worker_south_first[i
        ]);
117     }
118
119     MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
120     MPI_Barrier(MPI_COMM_WORLD);
121
122     if (rank == 0) {
123         master_entry();
124     } else if (rank > worker_proc_num) {
125         recovery_entry();
126     } else {
127         worker_entry();
128     }
129
130     MPI_Finalize();
131     return 0;
132 }
133
134
135 /**************************
136  * PROCESS ENTRY FUNCTIONS *
```

```
137  **************************/
138
139 void master_entry() {
140     int unfinished_workers = worker_proc_num;
141     int test[worker_proc_num];
142     MPI_Request test_request[worker_proc_num];
143     MPI_Status test_status;
144
145     // Non-blocking receives from all workers
146     for (int i = 0; i < worker_proc_num; i++) {
147         MPI_Irecv(&test[i], 1, MPI_INT, worker_rank[i], WORKER_FINISH_TAG, MPI_COMM_WORLD, &
    test_request[i]);
148     }
149
150     while (unfinished_workers) {
151         debug_printf("From %d (master) - unfinished workers: %d\n", rank, unfinished_workers);
152         int idx = -1;
153
154         // This operation completes if any worker is dead or finished
155         MPI_Waitany(worker_proc_num, test_request, &idx, &test_status);
156         debug_printf("From %d (master) - got message from %d: SOURCE: %d, TAG: %d, ERROR: %d\n",
157                      rank, worker_rank[idx], test_status.MPI_SOURCE, test_status.MPI_TAG,
    test_status.MPI_ERROR);
158
159         if (test_status.MPI_SOURCE != worker_rank[idx]) {
160             // Found dead worker - try to recover
161             int dead_rank = worker_rank[idx];
162             debug_printf("From %d (master) - found dead process %d\n", rank, dead_rank);
163
164             if (recovery_proc_num == 0) {
165                 printf("From %d (master) - no recovery processes, abort", rank);
166                 MPI_Abort(MPI_COMM_WORLD, RECOVERY_IMPOSSIBLE);
167                 return;
168             }
169
170             int recovery_proc_rank = size - recovery_proc_num;
171             debug_printf("From %d (master) - recovering dead process with %d\n", rank,
    recovery_proc_rank);
172             worker_rank[idx] = recovery_proc_rank;
173
174             debug_printf("From %d (master) - sending recovery data to %d\n", rank,
    recovery_proc_rank);
175             int err, tmp;
176             MPI_Status tmp_status;
177             err = MPI_Send(&dead_rank, 1, MPI_INT, recovery_proc_rank, RECOVERY_REQ_TAG,
    MPI_COMM_WORLD);
```

```
178             if (!err) err = MPI_Send(&worker_north_rank[idx], 1, MPI_INT, recovery_proc_rank,
    RECOVERY_REQ_TAG, MPI_COMM_WORLD);
179             if (!err) err = MPI_Send(&worker_south_rank[idx], 1, MPI_INT, recovery_proc_rank,
    RECOVERY_REQ_TAG, MPI_COMM_WORLD);
180             if (!err) err = MPI_Send(&worker_south_first[idx], 1, MPI_INT, recovery_proc_rank,
    RECOVERY_REQ_TAG, MPI_COMM_WORLD);
181             if (!err) err = MPI_Recv(&tmp, 1, MPI_INT, recovery_proc_rank, RECOVERY_REQ_TAG,
    MPI_COMM_WORLD, &tmp_status);
182
183             if (err) {
184                 printf("From %d (master) - failed to recover process\n", rank);
185                 MPI_Abort(MPI_COMM_WORLD, RECOVERY_FAILED);
186                 return;
187             }
188             recovery_proc_num--;
189             debug_printf("From %d (master) - successful recovery\n", rank);
190
191             MPI_Request send_req[2];
192             MPI_Status send_status[2];
193             int count = 0;
194
195             // Send recovered process rank to neighbours
196             debug_printf("From %d (master) - sending data to %d and %d\n",
197                          rank, worker_north_rank[idx], worker_south_rank[idx]);
198             if (worker_north_rank[idx] != NULL_WORKER) {
199                 MPI_Isend(&recovery_proc_rank, 1, MPI_INT, worker_north_rank[idx], RECOVERY_REQ_TAG,
     MPI_COMM_WORLD, &send_req[count]);
200                 worker_south_rank[worker_north_rank[idx]] = recovery_proc_rank;
201                 count++;
202             }
203             if (worker_south_rank[idx] != NULL_WORKER) {
204                 MPI_Isend(&recovery_proc_rank, 1, MPI_INT, worker_south_rank[idx], RECOVERY_REQ_TAG,
     MPI_COMM_WORLD, &send_req[count]);
205                 worker_north_rank[worker_south_rank[idx]] = recovery_proc_rank;
206                 count++;
207             }
208
209             // Wait for neighbours to receive recovered process rank
210             MPI_Waitall(count, send_req, send_status);
211             if ((count >= 1 && send_status[0].MPI_ERROR) || (count >= 2 && send_status[1].MPI_ERROR)
    ) {
212                 printf("From %d (master) - failed to send recovered rank\n", rank);
213                 MPI_Abort(MPI_COMM_WORLD, RECOVERY_FAILED);
214                 return;
215             }
216             debug_printf("From %d (master) - finished sending\n", rank);
217
```

```
218             // Non-blocking receive from new process
219             MPI_Irecv(&test[idx], 1, MPI_INT, worker_rank[idx], WORKER_FINISH_TAG, MPI_COMM_WORLD, &
    test_request[idx]);
220         } else {
221             // Process is not dead - it finished computing and now is waiting to send its data
222             printf("From %d (master) - found finished process %d\n", rank, worker_rank[idx]);
223             unfinished_workers--;
224         }
225     }
226     debug_printf("From %d (master) - all workers finished\n", rank);
227 
228     debug_printf("From %d (master) - stopping recovery processes\n", rank);
229     for (int i = 1; i <= recovery_proc_num; i++) {
230         int tmp = NULL_RANK;
231         debug_printf("From %d (master) - stop recovery proc %d\n", rank, size - i);
232         MPI_Send(&tmp, 1, MPI_INT, size - i, RECOVERY_REQ_TAG, MPI_COMM_WORLD);
233     }
234 
235     debug_printf("From %d (master) - receiving data from workers\n", rank);
236     eps = 0.;
237     for (int i = 0; i < worker_proc_num; i++) {
238         MPI_Status status;
239         double local_eps;
240         debug_printf("From %d (master) - receiving data from %d\n", rank, worker_rank[i]);
241         MPI_Recv(&start_row, 1, MPI_INT, worker_rank[i], WORKER_FINISH_TAG, MPI_COMM_WORLD, &status)
    ;
242         MPI_Recv(&last_row, 1, MPI_INT, worker_rank[i], WORKER_FINISH_TAG, MPI_COMM_WORLD, &status);
243         MPI_Recv(&A[start_row][0][0], (last_row - start_row + 1) * N * N, MPI_DOUBLE, worker_rank[i
    ], WORKER_FINISH_TAG, MPI_COMM_WORLD, &status);
244         MPI_Recv(&local_eps, 1, MPI_DOUBLE, worker_rank[i], WORKER_FINISH_TAG, MPI_COMM_WORLD, &
    status);
245         eps = Max(eps, local_eps);
246         debug_printf("From %d (master) - received data\n", rank);
247     }
248     start_row = 0;
249     last_row = N - 1;
250 
251     debug_printf("From %d (master) - finished\n", rank);
252     show_result();
253 }
254 
255 void recovery_entry() {
256     MPI_Status status;
257     int dead_rank;
258     debug_printf("From %d (recovery) - waiting request\n", rank);
259     MPI_Recv(&dead_rank, 1, MPI_INT, 0, RECOVERY_REQ_TAG, MPI_COMM_WORLD, &status);
260     if (dead_rank == NULL_RANK) {
```

```
261         debug_printf("From %d (recovery) - gracefully stopping\n", rank);
262         return;
263     }
264     debug_printf("From %d (recovery) - got request to recover %d\n", rank, dead_rank);
265 
266     MPI_Recv(&north_rank, 1, MPI_INT, 0, RECOVERY_REQ_TAG, MPI_COMM_WORLD, &status);
267     MPI_Recv(&south_rank, 1, MPI_INT, 0, RECOVERY_REQ_TAG, MPI_COMM_WORLD, &status);
268     MPI_Recv(&south_first, 1, MPI_INT, 0, RECOVERY_REQ_TAG, MPI_COMM_WORLD, &status);
269     debug_printf("From %d (recovery) - received recovery data: north: %d, south: %d, south first: %d
    \n",
270                 rank, north_rank, south_rank, south_first);
271 
272     load_worker_checkpoint(dead_rank);
273 
274     debug_printf("From %d (recovery) - successful loading, ack to master\n", rank);
275     int tmp;
276     MPI_Send(&tmp, 1, MPI_INT, 0, RECOVERY_REQ_TAG, MPI_COMM_WORLD);
277 
278     debug_printf("From %d (recovery) - entering worker loop\n", rank);
279     worker_state();
280 }
281 
282 void worker_entry() {
283     worker_init();
284     save_worker_checkpoint();
285     debug_printf("From %d (worker) - saved first checkpoint\n", rank);
286     worker_state();
287 }
288 
289 
290 /*****************************
291  * WORKER PROCESS FUNCTIONS *
292  *****************************/
293 
294 void worker_init() {
295     north_rank = worker_north_rank[rank - 1];
296     south_rank = worker_south_rank[rank - 1];
297     south_first = worker_south_first[rank - 1];
298 
299     free_workers_info();
300 
301     int num_rows = (N - 2) / worker_proc_num;
302     start_row = num_rows * (rank - 1) + 1;
303     last_row = start_row + num_rows - 1;
304     last_row += rank == worker_proc_num ? (N - 2) % worker_proc_num : 0;
305 
306     it_num = 0;
```

```
307     state = PENDING_COMPUTE;
308
309     matrix_init();
310     debug_printf("From %d (worker) - initialized, start row: %d, last_row: %d\n", rank, start_row,
        last_row);
311 }
312
313 void worker_state() {
314     for (; it_num < MAX_ITERATIONS; it_num++) {
315         debug_printf("From %d (worker) - it_num: %d, state: %d\n", rank, it_num, state);
316         switch (state) {
317             case PENDING_COMPUTE:
318                 suicide
319                 compute();
320                 state = PENDING_FIRST_SYNC;
321                 save_worker_checkpoint();
322                 debug_printf("From %d (worker) - saved checkpoint\n", rank);
323
324             case PENDING_FIRST_SYNC:
325                 suicide
326                 worker_sync(1);
327                 state = PENDING_SECOND_SYNC;
328                 save_worker_checkpoint();
329                 debug_printf("From %d (worker) - saved checkpoint\n", rank);
330
331             case PENDING_SECOND_SYNC:
332                 suicide
333                 worker_sync(0);
334                 state = PENDING_COMPUTE;
335                 save_worker_checkpoint();
336                 debug_printf("From %d (worker) - saved checkpoint\n", rank);
337         }
338     }
339
340     int tmp;
341     debug_printf("From %d (worker) - ack to master\n", rank);
342     MPI_Send(&tmp, 1, MPI_INT, 0, WORKER_FINISH_TAG, MPI_COMM_WORLD);
343
344     debug_printf("From %d (worker) - sending data to master\n", rank);
345     MPI_Send(&start_row, 1, MPI_INT, 0, WORKER_FINISH_TAG, MPI_COMM_WORLD);
346     MPI_Send(&last_row, 1, MPI_INT, 0, WORKER_FINISH_TAG, MPI_COMM_WORLD);
347     MPI_Send(&A[start_row][0][0], (last_row - start_row + 1) * N * N, MPI_DOUBLE, 0,
        WORKER_FINISH_TAG, MPI_COMM_WORLD);
348     MPI_Send(&eps, 1, MPI_DOUBLE, 0, WORKER_FINISH_TAG, MPI_COMM_WORLD);
349     debug_printf("From %d (worker) - finished\n", rank);
350 }
351
```

```
352 void worker_sync(int first_sync) {
353     while (1) {
354         MPI_Status status;
355         int dest = south_first == first_sync ? south_rank : north_rank;
356         debug_printf("From %d (worker) - entering sync with %d\n", rank, dest);
357         void *sendbuf = south_first == first_sync ? &A[last_row][0][0] : &A[start_row][0][0];
358         void *recvbuf = south_first == first_sync ? &A[last_row + 1][0][0] : &A[start_row -
    1][0][0];
359 
360         if (dest != NULL_RANK) {
361             int err = MPI_Sendrecv(sendbuf, N * N, MPI_DOUBLE, dest, first_sync ? FIRST_SYNC_TAG :
    SECOND_SYNC_TAG,
362                                    recvbuf, N * N, MPI_DOUBLE, dest, first_sync ? FIRST_SYNC_TAG :
    SECOND_SYNC_TAG, MPI_COMM_WORLD, &status);
363             if (err) {
364                 printf("From %d (worker) - Process %d appears to be dead\n", rank, dest);
365                 worker_recovery(dest, south_first == first_sync);
366                 continue;
367             }
368             debug_printf("From %d (worker) - successful sync with %d\n", rank, dest);
369         }
370         break;
371     }
372 }
373 
374 void free_workers_info() {
375     if (worker_rank)        free(worker_rank);
376     if (worker_north_rank)  free(worker_north_rank);
377     if (worker_south_rank)  free(worker_south_rank);
378     if (worker_south_first) free(worker_south_first);
379 
380     worker_rank = NULL;
381     worker_north_rank = NULL;
382     worker_south_rank = NULL;
383     worker_south_first = NULL;
384 }
385 
386 
387 /*****************************
388  * WORKER RECOVERY FUNCTIONS *
389  *****************************/
390 
391 void save_worker_checkpoint() {
392     debug_printf("From %d (worker) - saving checkpoint: it_num: %d, state: %d\n", rank, it_num,
    state);
393     char path[100];
394     snprintf(path, 100, "CP/control_point_%d.bin", rank);
```

```
395
396     FILE *cp_file = fopen(path, "wb");
397
398     if (cp_file == NULL) {
399         printf("From %d (worker) - could not save checkpoint\n", rank);
400         raise(SIGTERM);
401         return;
402     }
403
404     fwrite(&start_row, sizeof(start_row), 1, cp_file);
405     fwrite(&last_row, sizeof(last_row), 1, cp_file);
406     fwrite(&it_num, sizeof(it_num), 1, cp_file);
407     fwrite(&state, sizeof(state), 1, cp_file);
408     fwrite(&eps, sizeof(eps), 1, cp_file);
409
410     int start = north_rank == NULL_RANK ? start_row : start_row - 1;
411     int end = south_rank == NULL_RANK ? last_row : last_row + 1;
412
413     fwrite(&A[start][0][0], sizeof(double), (end - start + 1) * N * N, cp_file);
414
415     fclose(cp_file);
416     sync();
417 }
418
419 void load_worker_checkpoint(int dead) {
420     debug_printf("From %d (recovery) - loading checkpoint of %d\n", rank, dead);
421     char path[100];
422     snprintf(path, 100, "CP/control_point_%d.bin", dead);
423
424     sync();
425     FILE *cp_file = fopen(path, "rb");
426
427     if (cp_file == NULL) {
428         printf("From %d (recovery) - could not load checkpoint\n", rank);
429         raise(SIGTERM);
430         return;
431     }
432
433     fread(&start_row, sizeof(start_row), 1, cp_file);
434     fread(&last_row, sizeof(last_row), 1, cp_file);
435     fread(&it_num, sizeof(it_num), 1, cp_file);
436     fread(&state, sizeof(state), 1, cp_file);
437     fread(&eps, sizeof(eps), 1, cp_file);
438
439     int start = north_rank == NULL_RANK ? start_row : start_row - 1;
440     int end = south_rank == NULL_RANK ? last_row : last_row + 1;
441
```

```
442     fread(&A[start][0][0], sizeof(double), (end - start + 1) * N * N, cp_file);
443
444     fclose(cp_file);
445 }
446
447 void worker_recovery(int dead, int south) {
448     int err = MPI_Send(&dead, 1, MPI_INT, 0, RECOVERY_REQ_TAG, MPI_COMM_WORLD);
449     if (err) {
450         printf("From %d (worker) - failed to send recovery request\n", rank);
451         MPI_Abort(MPI_COMM_WORLD, RECOVERY_FAILED);
452         return;
453     }
454     MPI_Status status;
455     MPI_Recv(south ? &south_rank : &north_rank, 1, MPI_INT, 0, RECOVERY_REQ_TAG, MPI_COMM_WORLD, &
        status);
456 }
457
458
459 /*****************************
460  * MATRIX OPERATION FUNCTIONS *
461  *****************************/
462
463 void matrix_init() {
464     for (int i = start_row - 1; i <= last_row + 1; i++) {
465         for (int j = 0; j <= N - 1; j++) {
466             for (int k = 0; k <= N - 1; k++) {
467                 if (i == 0 || i == N - 1 || j == 0 || j == N - 1 || k == 0 || k == N - 1) {
468                     A[i][j][k] = 0.;
469                 } else {
470                     A[i][j][k] = (4. + i + j + k);
471                 }
472             }
473         }
474     }
475 }
476
477 void compute() {
478     relax();
479     resid();
480 }
481
482 void relax() {
483     debug_printf("From %d (worker) - started relax\n", rank);
484     for (int i = start_row; i <= last_row; i++) {
485         for (int j = 1; j <= N - 2; j++) {
486             for (int k = 1; k <= N - 2; k++) {
487                 B[i][j][k] = (A[i - 1][j][k] + A[i + 1][j][k] + A[i][j - 1][k] +
```

```
488                                 A[i][j + 1][k] + A[i][j][k - 1] + A[i][j][k + 1]) / 6.;
489             }
490         }
491     }
492 }
493
494 void resid() {
495     int start_flag = start_row == 0 ? 1 : 0;
496     int last_flag = last_row == N - 1 ? 1 : 0;
497
498     start_row = start_flag ? start_row + 1 : start_row;
499     last_row = last_flag ? last_row - 1 : last_row;
500
501     debug_printf("From %d (worker) - started resid\n", rank);
502     eps = 0.;
503     for (int i = start_row; i <= last_row; i++) {
504         for (int j = 1; j <= N - 2; j++) {
505             for (int k = 1; k <= N - 2; k++) {
506                 double e;
507                 e = fabs(A[i][j][k] - B[i][j][k]);
508                 A[i][j][k] = B[i][j][k];
509                 eps = Max(eps, e);
510             }
511         }
512     }
513     debug_printf("From %d (worker) - resid eps: %lf\n", rank, eps);
514
515     start_row = start_flag ? start_row - 1 : start_row;
516     last_row = last_flag ? last_row + 1 : last_row;
517 }
518
519 void show_result() {
520     double s = 0.0;
521     for (int i = start_row; i <= last_row; i++) {
522         for (int j = 0; j <= N - 1; j++) {
523             for (int k = 0; k <= N - 1; k++) {
524                 s = s + A[i][j][k] * (i + 1) * (j + 1) * (k + 1) / (N * N * N);
525             }
526         }
527     }
528
529     printf("S = %lf\neps = %lf\n", s, eps);
530
531     FILE * res = fopen("result_ft.txt", "w");
532     for (int i = start_row; i <= last_row; i++) {
533         for (int j = 0; j <= N - 1; j++) {
534             for (int k = 0; k <= N - 1; k++) {
```

```
535                     fprintf(res, "%lf ", A[i][j][k]);
536                 }
537                 fprintf(res, "\n");
538             }
539             fprintf(res, "\n");
540         }
541         fclose(res);
542 }
```

Листинг 2. jac_3d_mpi_ft.c

3.3. Задание №2: исходная версия

```
1
2  #include <math.h>
3  #include <stdio.h>
4  #include <mpi.h>
5
6  #define DEBUG 1
7
8  #define FIRST_SYNC_TAG 1215
9  #define SECOND_SYNC_TAG 1216
10 #define FINISH_TAG 1218
11
12 #define N 34
13 #define MAX_ITERATIONS 100
14
15 #define Max(a,b) ((a)>(b)?(a):(b))
16 #define debug_m_printf if (DEBUG && !rank) printf
17 #define debug_printf if (DEBUG) printf
18
19
20 void sync_edges();
21
22 void matrix_init();
23 void compute();
24 void relax();
25 void resid();
26 void show_result();
27
28
29 int rank, size;
30 int start_row, last_row;
31
32 double A[N][N][N], B[N][N][N];
33 double eps;
34
35
36 int main(int argc, char **argv) {
37     MPI_Init(&argc, &argv);
38     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
39     MPI_Comm_size(MPI_COMM_WORLD, &size);
40
41     debug_m_printf("size: %d\n", size);
42
43     int num_rows = (N - 2) / size;
44     start_row = num_rows * rank + 1;
45     last_row = start_row + num_rows - 1;
```

```
46      last_row += (rank == size - 1) ? (N - 2) % size : 0;
47      debug_printf("rank: %d, startrow: %d, lastrow: %d\n", rank, start_row, last_row);
48
49      matrix_init();
50
51      for (int it_num = 0; it_num < MAX_ITERATIONS; it_num++) {
52          compute();
53          sync_edges();
54      }
55
56      if (rank == 0) {
57          debug_printf("Receiving data\n");
58          MPI_Comm_size(MPI_COMM_WORLD, &size);
59          for (int i = 1; i < size; i++) {
60              MPI_Status status;
61              double local_eps;
62              MPI_Recv(&start_row, 1, MPI_INT, i, FINISH_TAG, MPI_COMM_WORLD, &status);
63              MPI_Recv(&last_row, 1, MPI_INT, i, FINISH_TAG, MPI_COMM_WORLD, &status);
64              MPI_Recv(&A[start_row][0][0], (last_row - start_row + 1) * N * N, MPI_DOUBLE, i,
    FINISH_TAG, MPI_COMM_WORLD, &status);
65              MPI_Recv(&local_eps, 1, MPI_DOUBLE, i, FINISH_TAG, MPI_COMM_WORLD, &status);
66              eps = Max(eps, local_eps);
67          }
68          start_row = 0;
69          last_row = N - 1;
70
71          MPI_Barrier(MPI_COMM_WORLD);
72          debug_printf("Finished\n");
73          show_result();
74      } else {
75          debug_printf("Sending data from %d\n", rank);
76          MPI_Send(&start_row, 1, MPI_INT, 0, FINISH_TAG, MPI_COMM_WORLD);
77          MPI_Send(&last_row, 1, MPI_INT, 0, FINISH_TAG, MPI_COMM_WORLD);
78          MPI_Send(&A[start_row][0][0], (last_row - start_row + 1) * N * N, MPI_DOUBLE, 0, FINISH_TAG,
     MPI_COMM_WORLD);
79          MPI_Send(&eps, 1, MPI_DOUBLE, 0, FINISH_TAG, MPI_COMM_WORLD);
80          MPI_Barrier(MPI_COMM_WORLD);
81      }
82
83      MPI_Finalize();
84      return 0;
85  }
86
87  void matrix_init() {
88      for (int i = start_row - 1; i <= last_row + 1; i++) {
89          for (int j = 0; j <= N - 1; j++) {
90              for (int k = 0; k <= N - 1; k++) {
```

```
91                  if (i == 0 || i == N - 1 || j == 0 || j == N - 1 || k == 0 || k == N - 1) {
92                      A[i][j][k] = 0.;
93                  } else {
94                      A[i][j][k] = (4. + i + j + k);
95                  }
96              }
97          }
98      }
99  }
100
101 void compute() {
102     relax();
103     resid();
104 }
105
106 void relax() {
107     debug_m_printf("Started relax\n");
108     for (int i = start_row; i <= last_row; i++) {
109         for (int j = 1; j <= N - 2; j++) {
110             for (int k = 1; k <= N - 2; k++) {
111                 B[i][j][k] = (A[i - 1][j][k] + A[i + 1][j][k] + A[i][j - 1][k] +
112                               A[i][j + 1][k] + A[i][j][k - 1] + A[i][j][k + 1]) / 6.;
113             }
114         }
115     }
116 }
117
118 void resid() {
119     int start_flag = start_row == 0 ? 1 : 0;
120     int last_flag = last_row == N - 1 ? 1 : 0;
121
122     start_row = start_flag ? start_row + 1 : start_row;
123     last_row = last_flag ? last_row - 1 : last_row;
124
125     debug_m_printf("Started resid\n");
126     eps = 0.;
127     for (int i = start_row; i <= last_row; i++) {
128         for (int j = 1; j <= N - 2; j++) {
129             for (int k = 1; k <= N - 2; k++) {
130                 double e;
131                 e = fabs(A[i][j][k] - B[i][j][k]);
132                 A[i][j][k] = B[i][j][k];
133                 eps = Max(eps, e);
134             }
135         }
136     }
137     debug_m_printf("Resid eps: %lf\n", eps);
```

```
138
139     start_row = start_flag ? start_row - 1 : start_row;
140     last_row = last_flag ? last_row + 1 : last_row;
141 }
142
143 void show_result() {
144     double s = 0.0;
145     for (int i = start_row; i <= last_row; i++) {
146         for (int j = 0; j <= N - 1; j++) {
147             for (int k = 0; k <= N - 1; k++) {
148                 s = s + A[i][j][k] * (i + 1) * (j + 1) * (k + 1) / (N * N * N);
149             }
150         }
151     }
152
153     printf("S = %lf\neps = %lf\n", s, eps);
154
155     FILE * res = fopen("result_noft.txt", "w");
156     for (int i = start_row; i <= last_row; i++) {
157         for (int j = 0; j <= N - 1; j++) {
158             for (int k = 0; k <= N - 1; k++) {
159                 fprintf(res, "%lf ", A[i][j][k]);
160             }
161             fprintf(res, "\n");
162         }
163         fprintf(res, "\n");
164     }
165     fclose(res);
166 }
167
168 void sync_edges() {
169     MPI_Request request[4];
170     MPI_Status status[4];
171
172     MPI_Comm_size(MPI_COMM_WORLD, &size);
173     if (rank) {
174         MPI_Irecv(&A[start_row - 1][0][0], N * N, MPI_DOUBLE, rank - 1, FIRST_SYNC_TAG,
    MPI_COMM_WORLD, &request[0]);
175         MPI_Isend(&A[start_row][0][0], N * N, MPI_DOUBLE, rank - 1, SECOND_SYNC_TAG, MPI_COMM_WORLD,
     &request[1]);
176     }
177     if (rank != size - 1) {
178         MPI_Isend(&A[last_row][0][0], N * N, MPI_DOUBLE, rank + 1, FIRST_SYNC_TAG, MPI_COMM_WORLD, &
    request[2]);
179         MPI_Irecv(&A[last_row + 1][0][0], N * N, MPI_DOUBLE, rank + 1, SECOND_SYNC_TAG,
    MPI_COMM_WORLD, &request[3]);
180     }
```

```
181
182     int ll = 4, shift = 0;
183     if (!rank) {
184         ll -= 2;
185         shift = 2;
186     }
187     if (rank == size - 1) {
188         ll -= 2;
189     }
190     if (ll) {
191         MPI_Waitall(ll, &request[shift], status);
192     }
193 }
```

Листинг 3. jac_3d_mpi_noft.c