ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

КУРСОВОЙ ПРОЕКТ

по дисциплине “Параллельные вычислительные технологии”

на тему

Разработка параллельной MPI-программы реализации алгоритма Флойда

Выполнил студент Дьяченко Даниил Вадимович
Ф.И.О.

Группы ИВ-621

Работу принял _____ доцент д.т.н. М.Г. Курносов
подпись

Защищена _____ Оценка _____

Новосибирск – 2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Математические модели в виде графов широко используются при моделировании разнообразных явлений, процессов и систем. Как результат, многие теоретические и реальные прикладные задачи могут быть решены при помощи тех или иных процедур анализа графовых моделей. Среди множества этих процедур может быть выделен некоторый определенный набор типовых алгоритмов обработки графов.

Далее рассмотрю способ параллельной реализации на MPI алгоритма Флойда (Floyd) на графе на примере задачи поиска кратчайших путей между всеми парами пунктов назначения. Задача состоит в том, что для имеющегося графа G требуется найти минимальные длины путей между каждой парой вершин графа. В качестве практического примера можно привести задачу составления маршрута движения транспорта между различными городами при заданном расстоянии между населенными пунктами и другие подобные задачи.

Постановка задачи

П у с т ь G е с т ь г р а о ф б = е , с) п д е л н я е н к о о т п о р р и о г п о

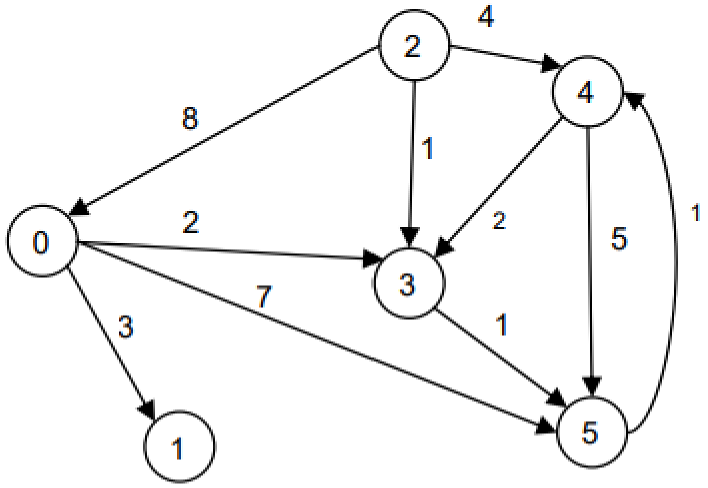

Рисунок 1 - Пример взвешенного ориентированного графа

Представление достаточно плотных графов, для которых почти все вершины соединены между собой дугами (т.е. $m \sim n^2$), может быть эффективно

$$a_{ij} \begin{cases} w\left(v_i, v_j\right), \text{если } (v_i, v_j) \in R \\ 0, \text{если } i = j \\ \infty, \text{иначе} \end{cases}$$

(для обозначения отсутствия ребра между вершинами в матрице смежности на соответствующей позиции используется знак бесконечности, при вычислениях знак бесконечности может быть заменен, например, на любое отрицательное число). Так, например, матрица смежности, соответствующая графу на рис. 1, приведена на рис. 2.

$$\begin{pmatrix} 0 & 3 & \infty & 2 & \infty & 7 \\ \infty & 0 & \infty & \infty & \infty & \infty \\ 8 & \infty & 0 & 1 & 4 & \infty \\ \infty & \infty & \infty & 0 & \infty & 1 \\ \infty & \infty & \infty & 2 & 0 & 5 \\ \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

Рисунок 2 - Матрица смежности для графа из рис. 1

Исходной информацией для задачи поиска кратчайших путей является взвешенный граф $G = (V, E)$ с одержащий n вершин.

Задача состоит в том, что для имеющегося графа G требуется найти минимальные длины путей между каждой парой вершин графа. В качестве практического примера можно привести задачу составления маршрута движения транспорта между различными городами при заданном расстоянии между населенными пунктами и другие подобные задачи.

2 Способ решения

Для поиска минимальных расстояний между всеми парами пунктов назначения Флойд предложил алгоритм, сложность которого имеет порядок n^3. В общем виде данный алгоритм может быть представлен следующим образом:

```
// Serial Floyd algorithm
for (k = 0; k < n; k++)
 for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
   A[i,j] = min(A[i,j],A[i,k]+A[k,j]);
```

(реализация операции выбора минимального значения min должна учитывать способ указания в матрице смежности несуществующих дуг графа). Как можно заметить, в ходе выполнения алгоритма матрица смежности A изменяется, после завершения вычислений в матрице A будет храниться требуемый результат – длины минимальных путей для каждой пары вершин исходного графа.

Как следует из общей схемы алгоритма Флойда, основная вычислительная нагрузка при решении задачи поиска кратчайших путей состоит в выполнении операции выбора минимальных значения. Данная операция является достаточно простой и ее распараллеливание не приведет к заметному ускорению вычислений. Более эффективный способ организации параллельных вычислений может состоять в одновременном выполнении нескольких операций обновления значений матрицы A.

Выполнение вычислений в подзадачах становится возможным только

т

о

г

д

а

,

к

Aik столбца *k* матрицы *A* должен быть передан всем подзадачам *i,k, 1≤i≤n* (см. рис. 3).

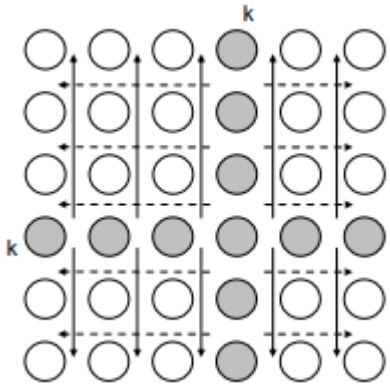

Рисунок 3 Информационная зависимость базовых подзадач (стрелками показаны направления обмена значениями на итерации k)

Как правило, число доступных процессоров p существенно меньше, чем число базовых задач n^2 ($p \ll n^2$). Возможный способ укрупнения вычислений состоит в использовании ленточной схемы разбиения матрицы A – такой подход соответствует объединению в рамках одной базовой подзадачи вычислений, связанных с обновлением элементов одной или нескольких строк.

3 Конфигурации оборудования

Количество узлов	Количество процессов на узел	Количество процессов всего
1	1	1
1	2	2
1	4	4
1	8	8
2	8	16
4	8	32
8	8	64

Таблица 1 – конфигурация узлов и процессов.

4 Результаты

Для эксперимента были выбраны следующие количества вершин: 100, 400,

Далее приведены результаты экспериментов. Полученное ускорение есть отношение времени выполнения последовательного алгоритма к времени выполнения параллельного алгоритма.

Количество процессов	Ускорение
1	1
2	1.526570048
4	1.858823529
8	1.915151515
16	1.389752205
32	1.271714066
64	0.717009751

Таблица 2 – результат эксперимента при 100 вершинах графа

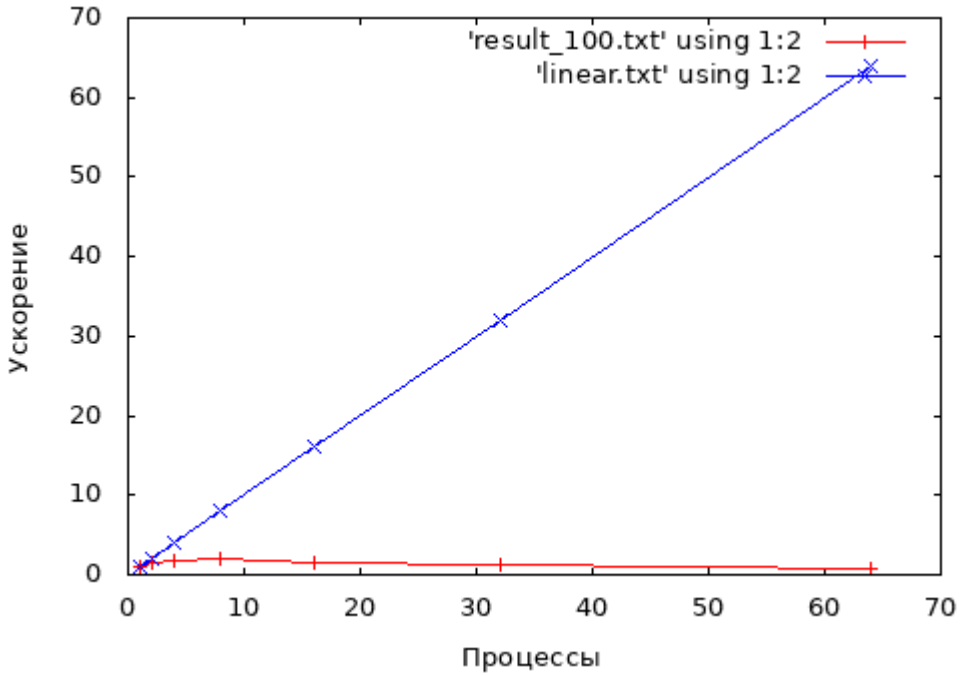

Рисунок 4 - результат эксперимента при 100 вершинах графа

Количество процессов	Ускорение
1	1
2	1.905673929
4	3.497363796
8	5.289036545
16	4.615830676
32	1.509591338
64	1.450088596

Таблица 3 – результат эксперимента при 400 вершинах графа

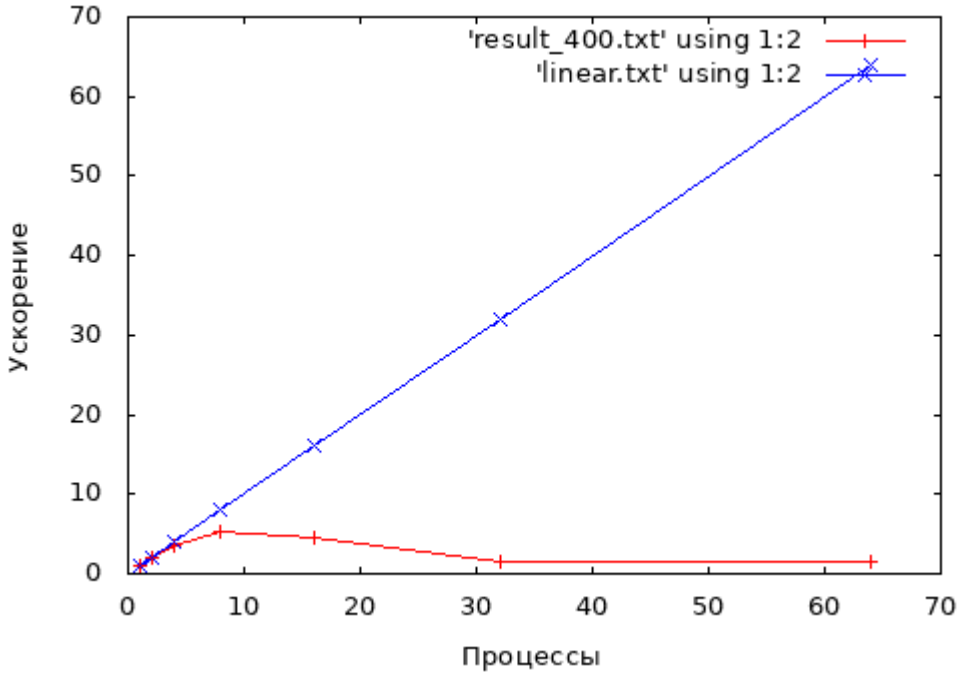

Рисунок 5- результат эксперимента при 400 вершинах графа

Количество процессов	Ускорение
1	1
2	2.069957071
4	4.100460951
8	8.029468621
16	12.756780286
32	16.333410741
64	18.825259176

Таблица 4 – результат эксперимента при 1600 вершинах графа

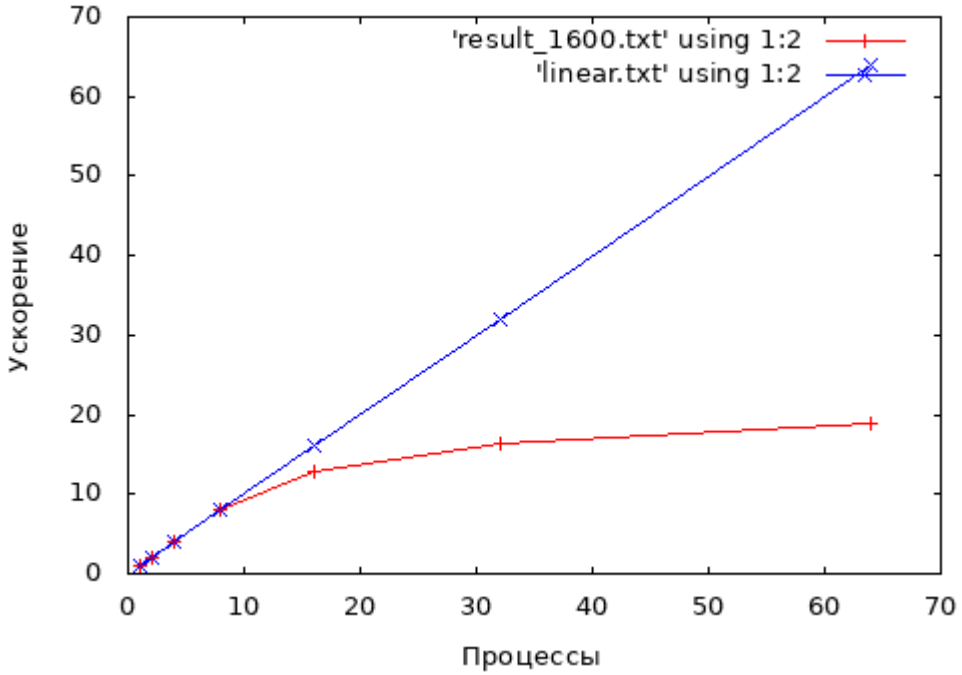

Рисунок 6 - результат эксперимента при 1600 вершинах графа

Количество процессов	Ускорение
1	1
2	1.394333926
4	1.115419639
8	2.114706752
16	4.089244011
32	7.319424121
64	46.772870629

Таблица 5 – результат эксперимента при 6400 вершинах графа

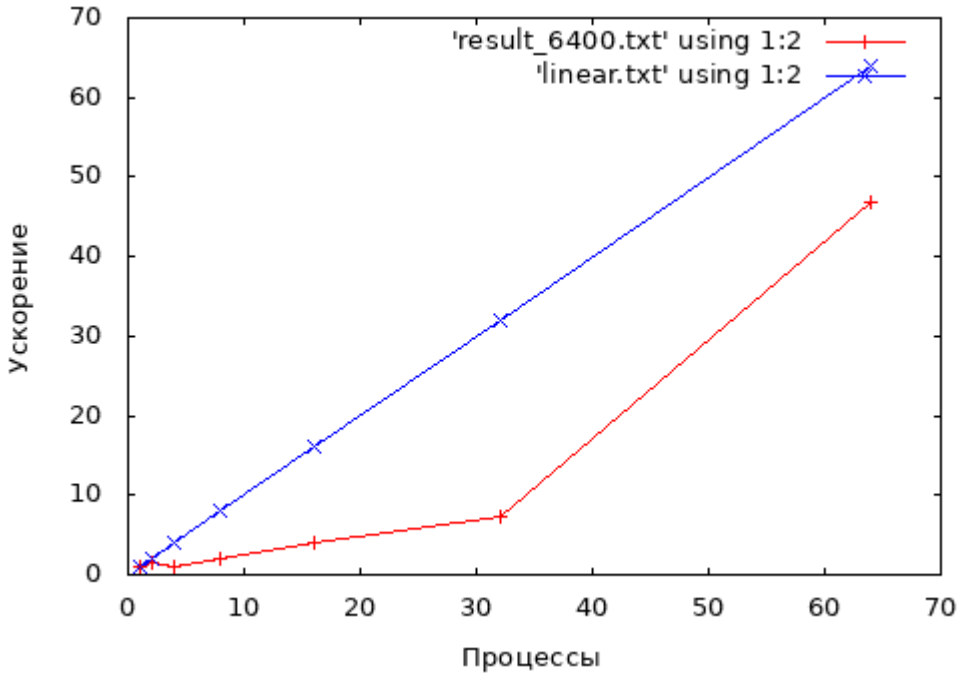

Рисунок 7 - результат эксперимента при 6400 вершинах графа

ЗАКЛЮЧЕНИЕ

В результате выполнения работы разработан и исследован алгоритм Флойда для поиска кратчайших путей на графе между всеми парами вершин.

Осуществлено моделирование разработанного алгоритма. Показано, что данный алгоритм хорошо подлежит распараллеливанию, так как имеет участки, которые можно выполнять независимо, не требующий больших затрат на пересылку данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Хорошевский В.Г. Архитектура вычислительных систем. – М.: МГТУ им. Н.Э. Баумана, 2008. – 520 с.
2. Евреинов Э.В., Хорошевский В.Г. Однородные вычислительные системы. – Новосибирск: Наука, 1978. – 320 с.
3. Rabenseifner R.. Automatic MPI Counter Profiling // Proceedings of the 42nd Cray User Group. – Noorwijk, The Netherlands, 2000. – 19 pp.
4. Han D., Jones T.. MPI Profiling // Technical Report UCRL-MI-209658 – Lawrence Livermore National Laboratory, USA, 2004. – 15 pp.
5. Thakur R., Rabenseifner R., and Gropp W. Optimization of collective communication operations in MPICH // Int. Journal of High Performance Computing Applications. – 2005. – Vol. 19, No. 1. – P. 49-66.
6. Balaji P., Buntinas D., Goodell D., Gropp W., Kumar S., Lusk E., Thakur R. and Traff J. L. MPI on a Million Processors // Proc. of the PVM/MPI – Berlin: Springer-Verlag, 2009. – P. 20-30.
7. Khoroshevsky V., Kurnosov M. Mapping Parallel Programs into Hierarchical Distributed Computer Systems // Proc. of “Software and Data Technologies”. – Sofia: INSTICC, 2009. – Vol. 2. – P. 123-128.

ПРИЛОЖЕНИЕ

1. Исходный код

```
/*
* main.c
*
*/
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <limits.h>
#include <time.h>

#define INF_PERC 30

static int rank;
static int commsize;

int min(int a, int b) {
    return (a < b) ? a : b;
}

int Min(int A, int B) {
    int Result = (A < B) ? A : B;

    if((A < 0) && (B >= 0)) Result = B;
    if((B < 0) && (A >= 0)) Result = A;
    if((A < 0) && (B < 0)) Result = -1;

    return Result;
}

void data_random_init(int *arr, int n) {
    srand(time(0));

    int rnd;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i != j) {
                rnd = rand() % 1000;
                if ((rnd % 100) < INF_PERC) {
                    arr[i * n + j] = INT_MAX;
                } else {
                    arr[i * n + j] = rnd + 1;
                }
            } else {
                arr[i * n + j] = 0;
            }
```

```
        }
    }
}

void dummy_data_init(int *arr, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            if (i == j) {
                arr[i * n + j] = 0;
            } else if (i == 0) {
                arr[i * n + j] = j;
            } else {
                arr[i * n + j] = INT_MAX;
            }
            arr[j * n + i] = arr[i * n + j];
        }
    }
}

void serial_Floyd(int *arr, int n) {
    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if ((arr[i * n + k] != INT_MAX) && (arr[k * n +
j] != INT_MAX)) {
                    arr[i * n + j] = min(arr[i * n + j], arr[i * n
+ k] + arr[k * n + j]);
                }
            }
        }
    }
}

void print_matrix(int *arr, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (arr[i * n + j] == INT_MAX) {
                printf("INF\t");
            } else {
                printf("%d\t", arr[i * n + j]);
            }
        }
        printf("\n");
    }
}

void par_print_matrix(int *arr, int n, int count_rows, int rank,
int commsize) {
    for (int p = 0; p < commsize; p++) {
        if (p == rank) {
            printf("rank = %d\n", rank);
```

```
            for (int i = 0; i < count_rows; i++) {
                for (int j = 0; j < n; j++) {
                    if (arr[i * n + j] == INT_MAX) {
                        printf("INF\t");
                    } else {
                        printf("%d\t", arr[i * n + j]);
                    }
                }
                printf("\n");
            }
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
}

void copy(int *start, int n, int *out) {
    if (!start || !out) {
        return;
    }

    for (int i = 0; i < n; i++) {
        out[i] = start[i];
    }
}

void row_distr(int *arr, int n, int k, int *row) {
    int num[commsize];
    int ind[commsize];
    int rem = n % commsize;
    int count_rows = n / commsize;

    for (int i = 0; i < commsize; i++) {
        num[i] = count_rows;

        if (rem > 0) {
            num[i]++;
            rem--;
        }

        ind[i] = (i > 0) ? ind[i - 1] + num[i - 1] : 0;

        #if 0
        if (rank == 0) {
            printf("k = %d [%d] num[%d] = %d\t", k, rank, i,
num[i]);
            printf("ind[%d] = %d\n", i, ind[i]);
        }
        #endif
    }

    int row_rank = -1;
```

```
    for (int i = 0; i < commsize; i++) {
        if (k < ind[i] + num[i]) {
            row_rank = i;
            break;
        }
    }

    #if 0
    printf("[%d] k = %d row_rank = %d\n", rank, k, row_rank);
    #endif

    if (row_rank == rank) {
        copy(&arr[(k - ind[rank]) * n], n, row);
    }

    MPI_Bcast(row, n, MPI_INT, row_rank, MPI_COMM_WORLD);
}

void par_Floyd(int *arr, int n, int count_rows) {
    int *row = calloc(n, sizeof(int));
    if (!row) {
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    for (int k = 0; k < n; k++) {
        row_distr(arr, n, k, row);

        #if 0
        for (int p = 0; p < commsize; p++) {
            if (p == rank) {
                printf("rank %d = ", rank);
                for (int i = 0; i < n; i++) {
                    if (row[i] == INT_MAX) {
                        printf("INF\t");
                    } else {
                        printf("%d\t", row[i]);
                    }
                }
                printf("\n");
            }
            MPI_Barrier(MPI_COMM_WORLD);
        }
        #endif

        for (int i = 0; i < count_rows; i++) {
            for (int j = 0; j < n; j++) {
                if ((arr[i * n + k] != INT_MAX) && (row[j] !=
INT_MAX)) {
                    arr[i * n + j] = min(arr[i * n + j], arr[i * n
+ k] + row[j]);
```

```
                }
            }
        }
    }

    free(row);
}

int compare(int *a, int *b, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (a[i * n + j] != b[i * n + j]) {
                return -1;
            }
        }
    }

    return 0;
}

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

    double t = 0;
    t -= MPI_Wtime();

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

    int n;
    int rem;
    int *arr;
    // int *cp_arr;
    int *recv_arr;
    int count_rows;
    int real_count_rows;

    if (rank == 0) {
        n = (argc > 1) ? atoi(argv[1]) : 0;
        if (n == 0) {
            printf("How to run:\nmpiexec ./main <number of
vertices>\n");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        } else if (n < commsize) {
            printf("Need number of vertices bigger then number of
processors\n");
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }

        arr = calloc(n * n, sizeof(int));
        if (!arr) {
```

```
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }

        data_random_init(arr, n);
        // dummy_data_init(arr, n);

        #if 0
        printf("arr:\n");
        print_matrix(arr, n);
        #endif
#if 0
        cp_arr = calloc(n * n, sizeof(int));
        if (!cp_arr) {
            MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
        }
        copy(arr, n * n, cp_arr);

        // printf("\nbefore floyd cp_arr:\n");
        // print_matrix(cp_arr, n);

        serial_Floyd(cp_arr, n);

        // printf("\nafter floyd cp_arr:\n");
        // print_matrix(cp_arr, n);
#endif
    }

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    real_count_rows = n / commsize;
    rem = n % commsize;

    if (rem > rank) {
        real_count_rows++;
    }

    count_rows = real_count_rows;

    #if 0
    printf("[%d] count_rows = %d\n", rank, count_rows);
    #endif

    recv_arr = calloc(n * count_rows, sizeof(int));
    if (!recv_arr) {
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    int *send_num = calloc(commsize, sizeof(int));
    int *send_ind = calloc(commsize, sizeof(int));
```

```
    if (!send_num || !send_ind) {
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    int old_rem = rem;
    #if 0
    if (rank == 0) {
        printf("rem = %d\n", rem);
    }
    #endif

    count_rows = n / commsize;

    for (int i = 0; i < commsize; i++) {
        send_num[i] = count_rows * n;

        if (rem > 0) {
            send_num[i] += n;
            rem--;
        }

        send_ind[i] = (i > 0) ? send_ind[i - 1] + send_num[i - 1] :
0;
        #if 0
        if (rank == 0) {
            printf("rem = %d\n", rem);
        }
        #endif
    }

    rem = old_rem;
    count_rows = real_count_rows;

    #if 0
    if (rank == 0) {
        for (int i = 0; i < commsize; i++) {
            printf("send_num[%d] = %d\n", i, send_num[i]);
            printf("send_ind[%d] = %d\n", i, send_ind[i]);
        }
    }
    #endif

    MPI_Scatterv(arr,  send_num,  send_ind,  MPI_INT,  recv_arr,
send_num[rank], MPI_INT, 0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);
    #if 0
    if (rank == 0) {
        printf("\nprint before par_Floyd\n");
    }
    par_print_matrix(recv_arr, n, count_rows, rank, commsize);
```

```
    printf("\n");
    #endif

    count_rows = real_count_rows;
    MPI_Barrier(MPI_COMM_WORLD);
    par_Floyd(recv_arr, n, count_rows);
    MPI_Barrier(MPI_COMM_WORLD);

    #if 0
    printf("[%d] FLOYD OK\n", rank);
    #endif

    #if 0
    if (rank == 0) {
        printf("\nprint after par_Floyd\n");
    }
    MPI_Barrier(MPI_COMM_WORLD);
    par_print_matrix(recv_arr, n, count_rows, rank, commsize);
    #endif

    int *recv_num = calloc(commsize, sizeof(int));
    int *recv_ind = calloc(commsize, sizeof(int));
    if (!recv_num || !recv_ind) {
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    #if 0
    if (rank == 0) {
        printf("rem = %d\n", rem);
    }
    #endif

    count_rows = n / commsize;

    for (int i = 0; i < commsize; i++) {
        recv_num[i] = count_rows * n;

        if (rem > 0) {
            recv_num[i] += n;
            rem--;
        }

        recv_ind[i] = (i > 0) ? recv_ind[i - 1] + recv_num[i - 1] :
0;
        #if 0
        if (rank == 0) {
            printf("rem = %d\n", rem);
        }
        #endif
    }
```

```
    count_rows = real_count_rows;

    #if 0
    if (rank == 0) {
        for (int i = 0; i < commsize; i++) {
            printf("recv_ind[%d] = %d\n", i, recv_ind[i]);
            printf("recv_num[%d] = %d\n", i, recv_num[i]);
        }
    }
    #endif

    #if 0
    if (rank == 0) {
        MPI_Gatherv(MPI_IN_PLACE, real_count_rows, MPI_INT, arr,
recv_num, recv_index, MPI_INT, 0, MPI_COMM_WORLD);
    } else {
        //  MPI_Gatherv(recv_arr,  recv_num[rank],  MPI_INT,  arr,
recv_num, recv_index, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Gatherv(recv_arr,  real_count_rows,  MPI_INT,  NULL,
NULL, NULL, MPI_INT, 0, MPI_COMM_WORLD);
    }
    #endif

    if (rank == 0) {
        for (int i = 0; i < count_rows; i++) {
            for (int j = 0; j < n; j++) {
                arr[i * n + j] = recv_arr[i * n + j];
            }
        }
        MPI_Status status;
        for (int i = 1; i < commsize; i++) {
            MPI_Recv(arr + recv_ind[i], recv_num[i], MPI_INT, i,
0, MPI_COMM_WORLD, &status);
        }
    } else {
        MPI_Send(recv_arr,  count_rows  *  n,  MPI_INT,  0,  0,
MPI_COMM_WORLD);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    #if 0
    if (rank == 0) {
        printf("\nprint ser after gather\n");
        print_matrix(arr, n);
    }
    #endif

    t += MPI_Wtime();
#if 0
    if (rank == 0) {
```

```
        if (compare(arr, cp_arr, n)) {
            printf("Compare is bad\n");
        } else {
            printf("Compare is good\n");
            printf("Elapsed time is %.5f sec\n", t);
        }
    }
#endif
    if (rank == 0) {
        printf("Elapsed time is %.5f sec\n", t);
    }

     MPI_Finalize();

    return 0;
    }
```