Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное
учреждение высшего образования
«Севастопольский государственный университет»

Кафедра информационных систем

КУРСОВОЙ ПРОЕКТ
ПО ДИСЦИПЛИНЕ «ТЕХНОЛОГИЯ РАСПРЕДЕЛЕННЫХ СИСТЕМ И ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ»
НА ТЕМУ:
«Параллельные вычисления»
Пояснительная записка

Листов 17

Студента 3 курса группы ИС/б-31-о
направление подготовки 09.03.02
*(подпись)*_____ Куркчи А. Э.
« » _____2017 г.
Руководитель _____
_____
(должность, ученое звание, фамилия и инициалы)
Оценка: _____

Члены комиссии _____
(подпись) (фамилия и инициалы)
_____
(подпись) (фамилия и инициалы)
_____
(подпись) (фамилия и инициалы)

г. Севастополь – 2017 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Данный курсовой проект посвящен исследованию параллельных вычислений.

Целью курсового проекта является закрепление, углубление, обобщение, а также применение на практике знаний по параллельным алгоритмам.

Курсовой проект связан с реализацией заданных алгоритмов по средством MPI, их тестированием и анализом полученных результатов.

Курсовой проект выполняется по индивидуальным заданиям и включает следующие задание:

- распараллеливание алгоритма сортировки;
- распараллеливание алгоритма работы с графами.

В данной работе проводится исследование алгоритма сортировки Шелла, алгоритма Беллмана-Форда, исходный код программы которого предоставлен в приложении Б.

1.АЛГОРИТМ СОРТИРОВКИ

В данной работе исследуется алгоритм сортировки Шелла.

Алгоритм сортировки Шелла является модификацией алгоритма «пузырьковой» сортировки с изменяющейся длиной шага.

Распараллеливание сортировки Шелла осуществляется за счет параллельного выполнения операции «Сравнить и разделить» над всеми парами блоков с варьирующимися расстояниями между блоками.

Текст программы приведен в приложении Б. Результаты выполнения программы приведены в приложении А.

1.1. Параллельный алгоритм на примере

Для объяснения работы параллельного алгоритма разберём его на примере гиперкуба начальной размерностью N = 2. Для этого случая воспользуемся случайным массивом на 16 элементов.

27	13	26	21	9	19	30	9	18	21	15	4	27	7	6	18

Выделим мастер процесс, им будет являться процесс с рангом 0, который будет распределять этот массив изначально. В данном случае каждый ПЭ в гиперкубе получит по 4 элемента массива, после чего произведёт сортировку этого блока последовательным алгоритмом Шелла. В результате после распределения и сортировки массив выглядит следующим образом:

ПЭ 0				ПЭ 1				ПЭ 2				ПЭ 3			
13	21	26	27	9	9	19	30	4	15	18	21	6	7	18	27

Далее на каждом шаге итерации, количество которых совпадает с размерностью гиперкуба, производится слияние блоков между двумя ПЭ таким образом, что на ПЭ с меньшим рангом остаётся блок размером в 2 раза больше исходного. При этом для пар ПЭ выполняется $i \; xor \; j = 1 \ll step$

В итоге на первой итерации выполняется обмен между ПЭ 0 и 1, 2 и 3.

ПЭ 0								ПЭ 2							
9	9	13	19	21	26	27	30	4	6	7	15	18	18	21	27

Реализовав сортировку слиянием между ПЭ, являющуюся частью операции «сравнить и разделить», переходим к следующей и последней итерации цикла, на которой выполнять эту операцию будут ПЭ с разницей во втором разряде двоичной записи их номера, а именно ПЭ 0 и 2.

ПЭ 0															
4	6	7	9	9	13	15	18	18	19	21	21	26	27	27	39

Таким образом после реализации N итераций операции «сравнить и разделить» между ПЭ находящихся на разном расстоянии на ПЭ 0, который изначально определён как мастер процесс, остаётся массив изначальной размерности, отсортированный параллельным алгоритмом Шелла.

Простая реализация этого алгоритма предполагает разбиение массива на равные изначально блоки и рассылку размера блока, а также самих блоков с применением операций MPI_Bcast и MPI_Scatter, для реализации сортировки массивов произвольного размера возможны два подхода:

1) Padding – создание отступа в конце или начале массива (дополнение до кратной размерности). Этот метод самый простой в программировании, достаточно при считывании или генерации массива запоминать максимальный элемент, после чего инкрементировать его, используя как значение для дополнения. Из итогового массива необходимо будет отсечь эти элементы, стоящие в итоге в конце.
2) Расчёт размеров блоков с последующей раздачей их операцией MPI_Scatter, а раздача блоков с применением операции MPI_Scatterv, позволяющей рассылать блоки разной длины.

1.2. Анализ результатов

Тестирование проводится с размерностью гиперкуба 1, 2 и 3. Результат представлен в виде графика. На нём обозначены затраты времени на сотню запусков алгоритма с 2, 4 и 8 сортирующими процессами.

Как можно наблюдать из рисунка 1.1 параллельный алгоритм выполняется быстрее последовательного и имеет тенденцию к увеличению эффективности по сравнению с последовательным алгоритмом.

Наиболее эффективным оказался случай при размерности гиперкуба 3. Однако при такой размерности требуется 8 сортирующих процессов и один управляющий, что в совокупности даёт 9 процессов, которые на 8-ми ядерном процессоре вынуждены испытывать ограничения связанные с размерностью кванта времени. Наиболее стабильным по времени оказался случай при размерности гиперкуба 2. Дальнейшее увеличение размерности гиперкуба привело к уменьшению эффективности параллельного алгоритма. Это объясняется увеличением накладных расходов при обмене данными между процессами.

2. АЛГОРИТМ НА ГРАФАХ

В данной работе будет исследоваться алгоритм Беллмана-Форда.

Алгоритм Беллмана-Форда — алгоритм поиска кратчайшего пути во взвешенном графе. За время $O(|V| \times |E|)$ алгоритм находит кратчайшие пути от одной вершины графа до всех остальных. Алгоритм Беллмана-Форда допускает рёбра с отрицательным весом.

Распараллеливание данного алгоритма осуществляется за счет одновременной “релаксации” по различным ребрам на соответствующих процессах. После чего все процессы производят минимизацию по полученным результатом. Итерация продолжается до тех пор пока продолжается “релаксация” ребер.

Текст программы приведен в приложении Б. Результаты выполнения программы приведены в приложении А.

2.1. Реализация параллельного алгоритма

Как сказано ранее, распараллеливание состоит в возможности параллельной «релаксации» по различным рёбрам, результаты минимизации на каждой итерации должны храниться на всех ПЭ. В связи с этим каждый ПЭ должен знать количество вершин, массив кратчайших расстояний до этих вершин и свою часть ребер графа. Простая параллельная реализация алгоритма Беллмана-Форма предполагает отсутствие отрицательных циклов.

Первоначальной инициализацией является рассылка количества вершин, рёбер и начальной вершины с применением операции MPI_Bcast. Далее каждый ПЭ получает с мастер-процесса свой сегмент матрицы смежности, а в конкретной реализации списка смежности (что является более оптимальным в общем случае), через операцию MPI_Scatterv. В текущей реализации принято

допущение, что число рёбер строго кратно количеству процессов, потому возможно использование операции MPI_Scatter, что упрощает задачу программирования системы. Дальнейшее обобщение алгоритма возможно с применением методов аналогичных методам для сортировки Шелла, приведённым в пункте 1.1 данного курсового проекта. Каждый ПЭ выполняет релаксацию кратчайших путей по данным ему ребрам графа, сохраняя при этом флаг была ли произведена релаксация хотя бы один раз за текущую итерацию.

В последствии происходит минимизация полученных кратчайших путей с использованием операции MPI_Allreduce по функции MPI_MIN. Эта операция позволяет выбрать для каждой вершины минимальный кратчайший путь среди тех, которые получились в результате релаксации на каждом процессе. Результаты такой минимизации значений остаются на каждом ПЭ, после чего производится проверка на необходимость дальнейшей релаксации. Если локально релаксация не производилась и итоговый массив расстояний совпадает с локальным цикл завершается и результат вычислений находится во всех ПЭ.

Сложность такого алгоритма падает до $O(|V| \times |E| / N)$, где N – количество процессорных элементов. При этом накладные расходы на пересылки и минимизации растут пропорционально количеству элементов.

2.2. Анализ результатов

Для тестирования используется следующие параметры графа: количество вершин, разрежённость графа (количество ребер), количество процессов.

Как видно из рисунков 2.1, 2.3 и 2.5 время алгоритма пропорционально как числу вершин, так и количеству ребер графа.

Из рисунков 2.2, 2.4, 2.6 видно, что параллельный алгоритм, как с увеличением вершин, так и с увеличением количества ребер имеет тенденцию к увеличению эффективности по сравнению с последовательным алгоритмом. Наиболее эффективным оказался случай при 4 процессах.

ЗАКЛЮЧЕНИЕ

В результате данного курсового проекта были закреплены и применены на практике знания по параллельным вычислениям.

Были исследованы и реализованы параллельные алгоритмы: алгоритм быстрой сортировки и алгоритма Беллмана-Форда.

По результатам тестирования данных алгоритмов можно сделать следующие выводы.

- параллельные алгоритмы позволяют повысить скорость выполнения программ приблизительно в два-три раза (в рамках данного исследования).
- в следствии накладных расходов (возникающих при обмене данными между процессами) при небольшой размерности задачи, параллельный алгоритм уступает последовательному
- при увеличении количество процессов, увеличивается количество накладных расходов, как следствие момент когда параллельный алгоритм будет эффективнее последовательного наступает позже
- по двум предыдущем утверждениям, можно сделать вывод, что для небольших вычислительных задач будет эффективнее использовать последовательный алгоритм или параллельный, с небольшим количество процессов

ПРИЛОЖЕНИЕ А

Ниже представлены результаты распараллеливания алгоритма сортировки Шелла при различной размерности гиперкуба.

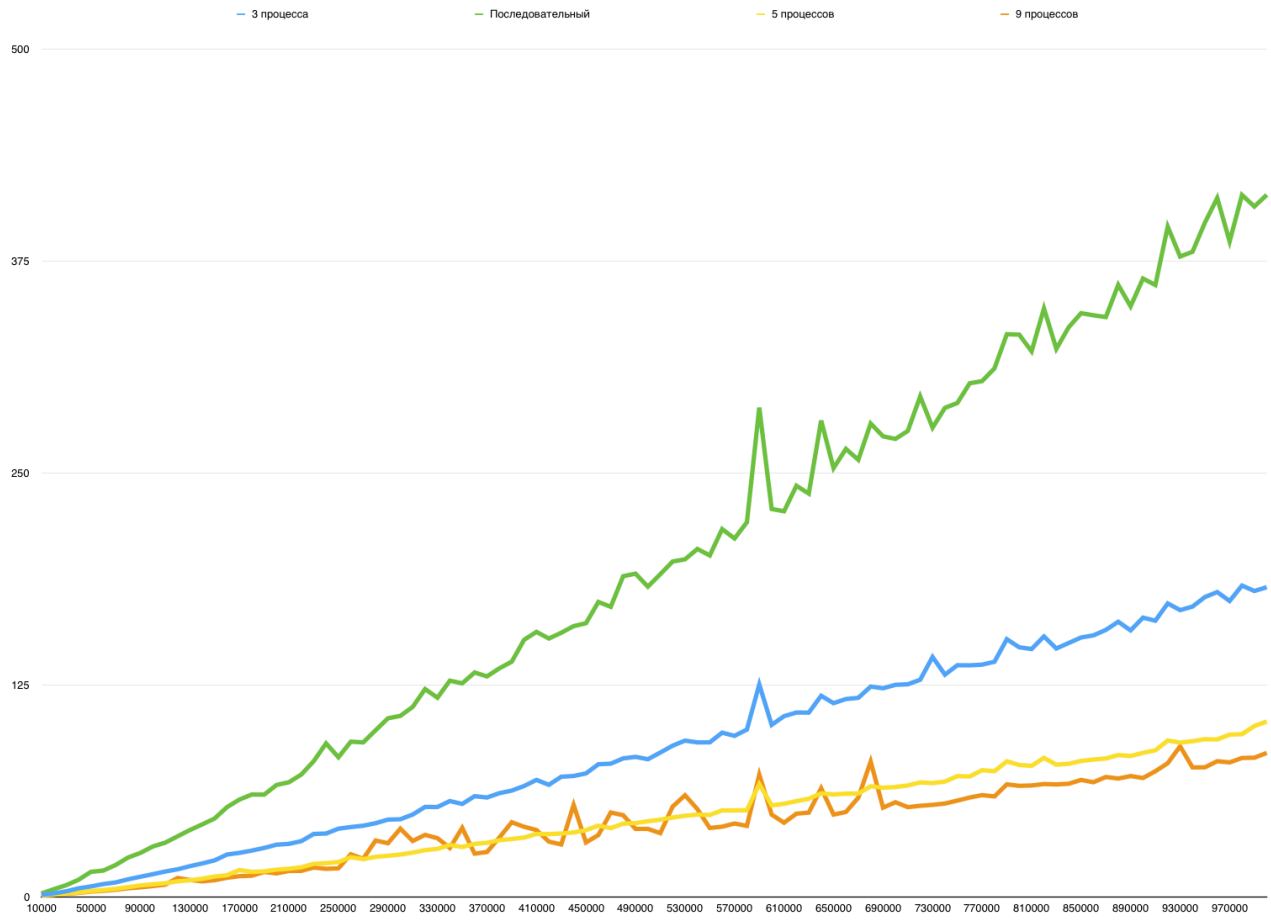

Рисунок 1.1 - Результат работы алгоритма при N = 1, 2, 3

Ниже представлены результаты распараллеливания алгоритма Беллмана-Форда при различном количестве вершин и разрежённости графа.

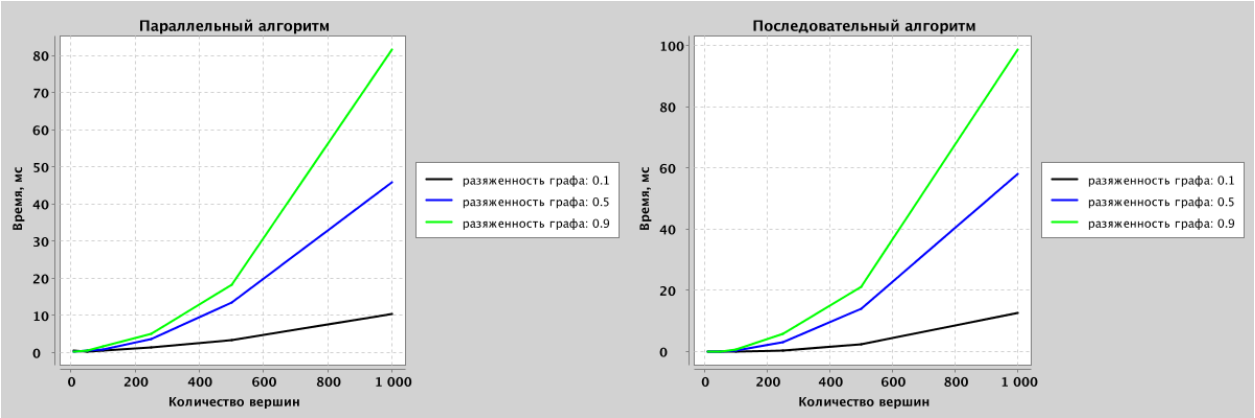

Рисунок 2.1 – Зависимость времени работы алгоритма при различной разрежённости графа (2 процесса)

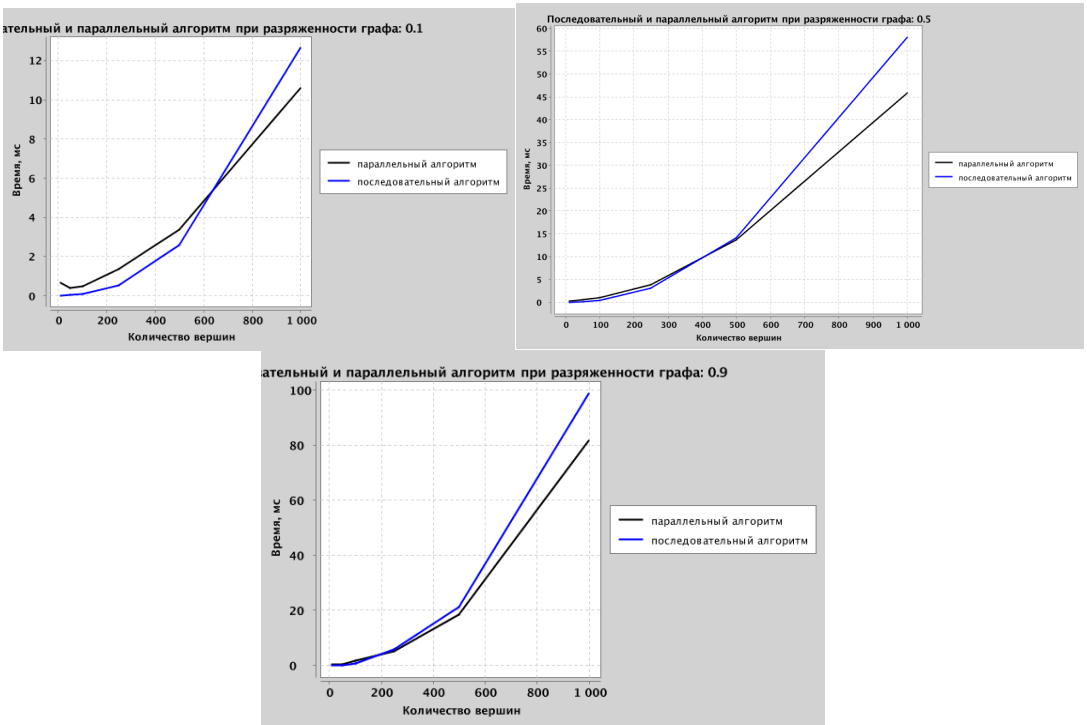

Рисунок 2.2 – Сравнение последовательного и параллельного алгоритмов (2 процесса)

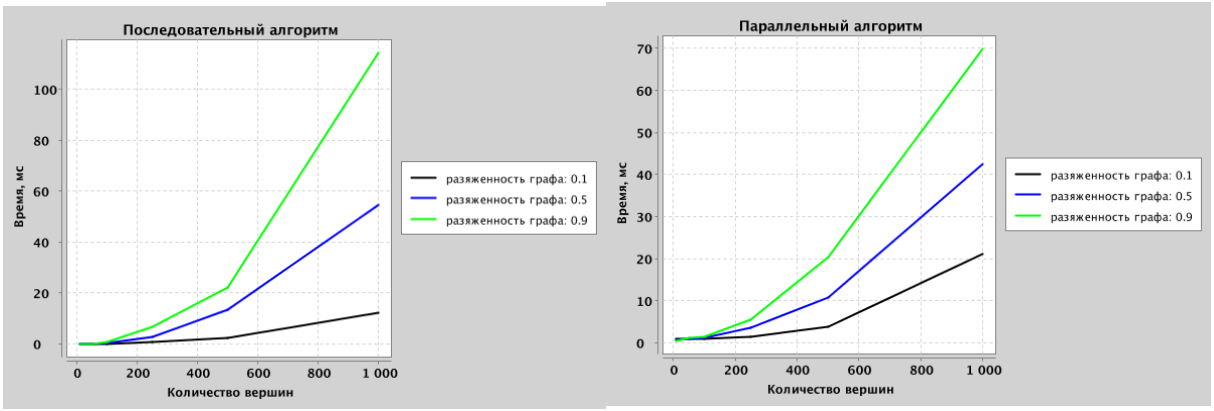

Рисунок 2.3 – Зависимость времени работы алгоритма при различной разрежённости графа (4 процесса)

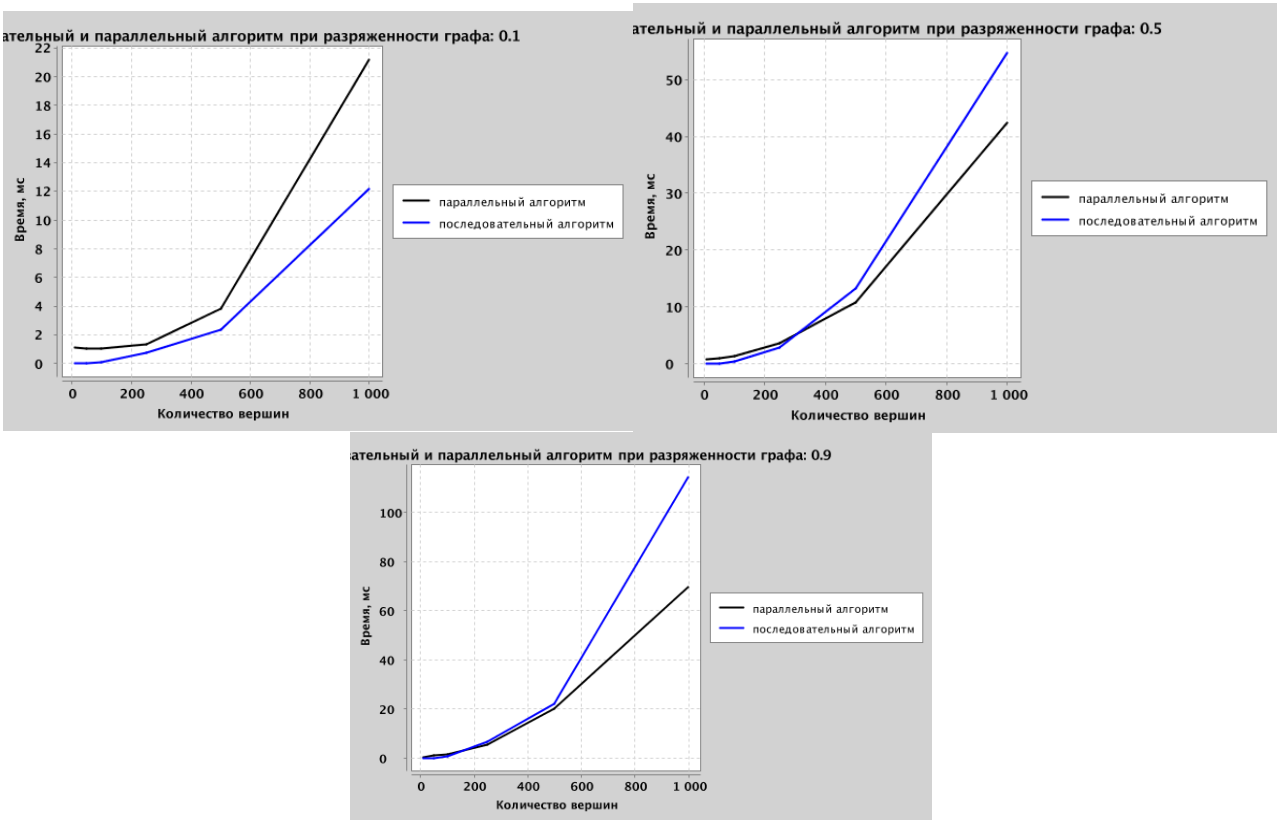

Рисунок 2.4 – Сравнение последовательного и параллельного алгоритмов (4 процесса)

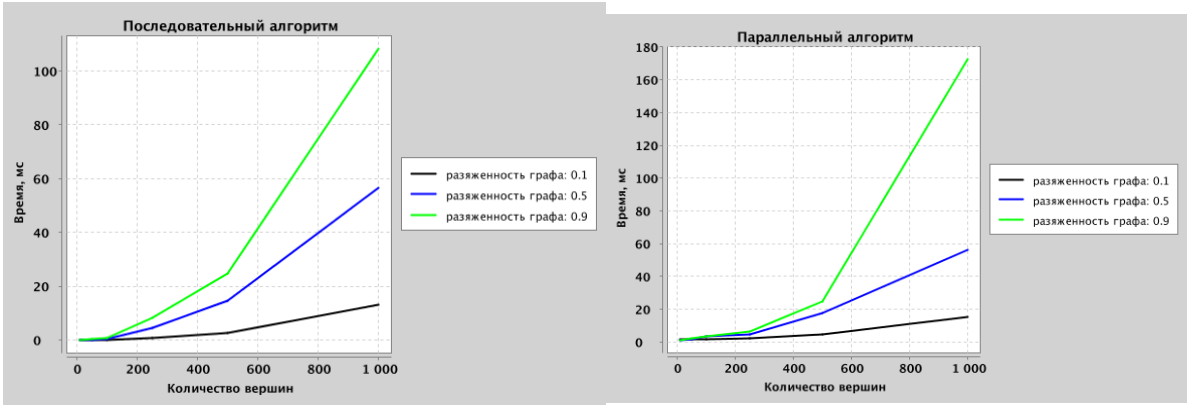

Рисунок 2.5 – Зависимость времени работы алгоритма при различной разрежённости графа (8 процессов)

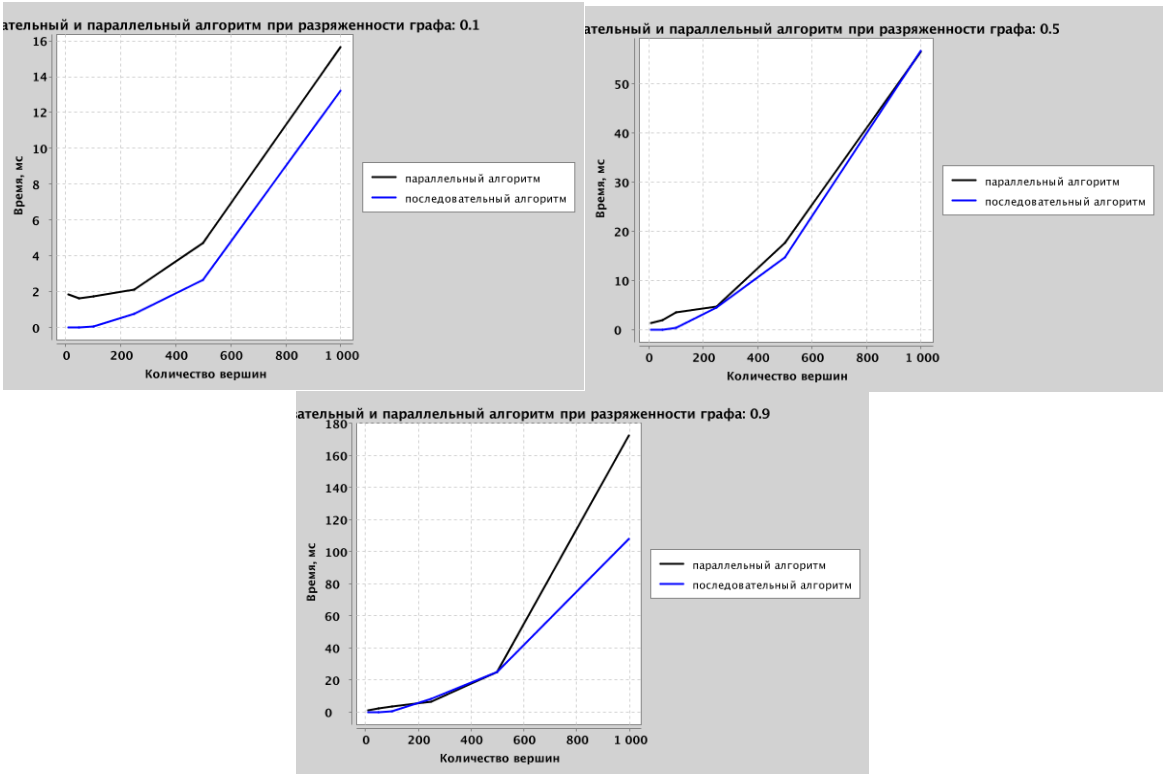

Рисунок 2.6 – Сравнение последовательного и параллельного алгоритмов (4 процессов)

ПРИЛОЖЕНИЕ Б

Исходный код программы параллельной сортировки и тестирования алгоритма Шелла

```
#include <iostream>
#include <mpi.h>
#include <cmath>
#include <iomanip>

#define ITERATIONS_COUNT 1
#define DEBUG false

#define OP_CLOSE 0
#define OP_SHELL 1

#define ROOT 0

#define VALUE_TYPE MPI_INTEGER
typedef int val_t;

using namespace std;
using namespace chrono;

unsigned int array_size;

int mpi_rank;
int mpi_size;
val_t *block;
int block_size;

MPI_Comm hyperComm;
MPI_Comm hyperCube;
int hyperCubeDims;
int hyperCubeSize;

void op(int id);

void free_mem(void *ptr);

void *vmem(const size_t size);

int *imem(const int size);

val_t *tmem(const int size);

long long
timer(void (*before)(void **), void (*timed_function)(void **), void (*after)(void **), void **arg, int iterations,
      long long *time_full);

val_t *merge(int *v1, int n1, int *v2, int n2);

const int *list(int i, int dims);

const int *ilist(int dims);


void shellSort(val_t *array, const int size) {
    val_t t;
    for (unsigned k = (unsigned int) (size / 2); k > 0; k /= 2) {
        for (unsigned i = k; i < size; i += 1) {
            t = array[i];
            unsigned j;
            for (j = i; j >= k; j -= k) {
                if (t < array[j - k]) {
                    array[j] = array[j - k];
                } else {
                    break;
                }
            }
            array[j] = t;
        }
    }
}

void shellSequential(void **arg) {
    val_t *array = *((val_t **) arg);
    shellSort(array, array_size);
}

void shellParallelCommon() {
    MPI_Status status;
    int step = 1;
    val_t *other;
```

```
80      while (step < hyperCubeSize) {
81          if (mpi_rank % (step * 2) == 0) {
82              if (mpi_rank + step < hyperCubeSize) {
83                  int second_block_size;
84                  MPI_Recv(&second_block_size, 1, MPI_INT, mpi_rank + step, 0, hyperCube, &status);
85                  other = tmem(second_block_size);
86                  MPI_Recv(other, second_block_size, MPI_INT, mpi_rank + step, 0, hyperCube, &status);
87                  block = merge(block, block_size, other, second_block_size);
88                  block_size += second_block_size;
89              }
90          } else {
91              int near = mpi_rank - step;
92              MPI_Send(&block_size, 1, MPI_INT, near, 0, hyperCube);
93              MPI_Send(block, block_size, MPI_INT, near, 0, hyperCube);
94              break;
95          }
96          step *= 2;
97      }
98  }
99
100 void shellParallelMaster(void **arg) {
101     val_t *array = *((val_t **) arg);
102
103     op(OP_SHELL);
104     block_size = array_size / hyperCubeSize;
105
106     MPI_Bcast(&block_size, 1, MPI_INT, ROOT, hyperCube);
107     block = tmem(block_size);
108     MPI_Scatter(array, block_size, VALUE_TYPE, block, block_size, VALUE_TYPE, ROOT, hyperCube);
109     shellSort(block, block_size);
110
111     shellParallelCommon();
112 }
113
114 void shellParallelSlave() {
115     MPI_Bcast(&block_size, 1, MPI_INT, ROOT, hyperCube);
116     block = tmem(block_size);
117     MPI_Scatter(NULL, 0, VALUE_TYPE, block, block_size, VALUE_TYPE, ROOT, hyperCube);
118     shellSort(block, block_size);
119
120     shellParallelCommon();
121 }
122
123 void preGenerateArray(void **arg) {
124     val_t *array = *((val_t **) arg);
125     for (int i = 0; i < array_size; i++) {
126         array[i] = rand();
127     }
128 }
129
130 void post(void **arg) {
131 }
132
133 long long test(void (*timed_function)(void **), long long *full_time) {
134     val_t *array = tmem(array_size);
135     long long time = timer(preGenerateArray, timed_function, post, (void **) &array, ITERATIONS_COUNT, full_time);
136     free_mem(array);
137     return time;
138 }
139
140 void init() {
141     MPI_Cart_create(hyperComm, hyperCubeDims, list(2, hyperCubeDims), list(0, hyperCubeDims), 0, &hyperCube);
142 }
143
144 int master() {
145     long long sequential_ns, parallel_ns, sequential_full, parallel_full;
146     srand((unsigned int) time(0));
147
148     for (array_size = (unsigned int) 10000;
149          array_size <= 1000000; array_size += 10000) {
150         sequential_ns = test(shellSequential, &sequential_full);
151         parallel_ns = test(shellParallelMaster, &parallel_full);
152 #if DEBUG
153         cout << "Sequential" << "\t" << sequential_ns << "ns = " << sequential_ns / 1000 << "mcs = "
154              << sequential_ns / 1000 / 1000 << "ms" << endl;
155         cout << "Parallel  " << "\t" << parallel_ns << "ns = " << parallel_ns / 1000 << "mcs = "
156              << parallel_ns / 1000 / 1000 << "ms" << endl;
157 #else
158         cout << setw(6) << array_size << "\t" << setw(10) << sequential_ns << "\t" << setw(10) << parallel_ns
159              << "\t" << setw(10) << ((parallel_ns - sequential_ns) / (sequential_ns * 1.0)) * 100 << "%"
160              //              << "\t" << setw(10) << ((parallel_full - sequential_ns) / (sequential_ns * 1.0)) * 100
<< "%"
161              << endl;
162 #endif
163     }
164     op(OP_CLOSE);
165     return 0;
166 }
167
168 int slave() {
169     int op;
170     while (true) {
171         MPI_Barrier(hyperCube);
172         MPI_Bcast(&op, 1, MPI_INTEGER, 0, hyperCube);
```

```
        switch (op) {
            case OP_SHELL:
                shellParallelSlave();
                break;
            case OP_CLOSE:
                return 0;
            default:
                return op;
        }
    }
}

int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);

    array_size = (unsigned int) (mpi_size * 8);

    hyperCubeDims = (int) floor(log2(mpi_size));
    hyperCubeSize = (int) pow(2, hyperCubeDims);

    MPI_Group worldGroup, hyperGroup;
    MPI_Comm_group(MPI_COMM_WORLD, &worldGroup);
    MPI_Group_incl(worldGroup, hyperCubeSize, ilist(hyperCubeSize), &hyperGroup);
    MPI_Comm_create(MPI_COMM_WORLD, hyperGroup, &hyperComm);

    int code = 0;
    if (mpi_rank < hyperCubeSize) {
        init();
        code = mpi_rank ? slave() : master();
    }

    MPI_Finalize();
    return code;
}

void op(int id) {
    MPI_Barrier(hyperCube);
    MPI_Bcast(&id, 1, MPI_INTEGER, ROOT, hyperCube);
}

void free_mem(void *ptr) {
    if (ptr != NULL) {
        try {
            free(ptr);
        } catch (...) {

        }
    }
}

void *vmem(const size_t size) {
    return malloc(size);
}

int *imem(const int size) {
    return (int *) vmem(sizeof(int) * size);
}

val_t *tmem(const int size) {
    return (val_t *) vmem(sizeof(val_t) * size);
}

long long
timer(void (*before)(void **), void (*timed_function)(void **), void (*after)(void **), void **arg, int iterations,
      long long *time_full) {
    long long time_only = 0;
    if (time_full != NULL) {
        (*time_full) = 0;
    }
    for (int i = 0; i < iterations; i++) {
        auto before_time = chrono::high_resolution_clock::now();
        before(arg);
        auto start_time = high_resolution_clock::now();
        timed_function(arg);
        auto end_time = high_resolution_clock::now();
        after(arg);
        auto after_time = high_resolution_clock::now();

        time_only += duration_cast<nanoseconds>(end_time - start_time).count();
#if DEBUG
        cout << i << ": " << duration_cast<milliseconds>(end_time - start_time).count() << "ms" << endl;
#endif
        if (time_full != NULL) {
            (*time_full) += duration_cast<nanoseconds>(after_time - before_time).count();
        }
    }

    return time_only;
}

val_t *merge(int *v1, int n1, int *v2, int n2) {
    int i = 0, j = 0, k = 0,
```

```
267             *result;
268
269     result = tmem(n1 + n2);
270
271     while (i < n1 && j < n2)
272         if (v1[i] < v2[j]) {
273             result[k++] = v1[i++];
274         } else {
275             result[k++] = v2[j++];
276         }
277     if (i == n1) {
278         while (j < n2) {
279             result[k++] = v2[j++];
280         }
281     } else {
282         while (i < n1) {
283             result[k++] = v1[i++];
284         }
285     }
286     return result;
287 }
288
289 const int *list(int i, int dims) {
290     int *res = imem(dims);
291     for (int k = 0; k < dims; res[k++] = i);
292     return res;
293 }
294
295 const int *ilist(int dims) {
296     int *res = imem(dims);
297     for (int k = 0; k < dims; res[k] = k, k++);
298     return res;
299 }
```

Так как исходный код программы для алгоритма Беллмана-Форда слишком велик он доступен по следующей ссылке: https://github.com/justnero-ru/university/tree/master/semestr.06/TPCиПВ/bellman