МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет информационных технологий

Кафедра параллельных вычислений

ОТЧЕТ

О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ

«Распараллеливание итерационного алгоритма решения системы линейных алгебраических уравнений средствами MPI»

студента 2 курса, группы 20205

Муратова Максима Александровича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
доцент
Власенко А. Ю.

Новосибирск, 2022

СОДЕРЖАНИЕ

ЦЕЛИ

1. Распараллелить программу разными способами;
2. Найти зависимость ускорения программы от числа процессов, исполнявших её;
3. При помощи профилировщика выяснить, сколько времени потребовалось выполнение вычислительной части программы, а сколько – на операции MPI;
4. Регулированием количеством потоков выявить, какой способ распараллеливания эффективнее.

ЗАДАНИЕ

1. Написать 3 версии программы: последовательную, параллельную с дублированием векторов b и x и разрезанием матрицы по строкам, параллельную с разрезанием матрицы по столбцам и векторов;
2. Алгоритм решения СЛАУ — метод сопряжённых градиентов. Пусть нам необходимо решить систему линейных уравнений $Ax=b$, где A – матрица коэффициентов, b – свободные члены, x – искомый вектор решения, $\varepsilon>0$ – погрешность решения.
   Пусть x_0 – первое приближение к решению x, в моей реализации x_0 состоит из единиц. $r_0=b-Ax_0$ – остаток, $z_0=r_0$, α_i,β_i – коэффициенты.
   Одна итерация данного итерационного алгоритма состоит из следующих операций:
   $$r_{n+1}=r_n-\alpha_{n+1}Az_n$$
   $$z_{n+1}=r_{n+1}+\beta_{n+1}z_n$$
   $$\alpha_{n+1}=\frac{(r_n,r_n)}{(Az_n,z_n)}$$
   $$\beta_{n+1}=\frac{(r_{n+1},r_{n+1})}{(r_n,r_n)}$$
   $$x_{n+1}=x_n+a_{n+1}z_n$$
   Условие выхода из цикла – $\frac{\|r_n\|}{\|b\|}<\varepsilon$.
   (a,b) – скалярное произведение в декартовых координатах, $\|a\|=\sqrt{(a,a)}$ – норма вектора.
3. Профилировщик – Intel Trace Analyzer and Collector (ITAC).

ОПИСАНИЕ РАБОТЫ

Файлы с исходными кодами каждых версий программы находятся в отдельных папках. Исходные коды первой программы в папке `series`, второй – `paral`, третьей – `split`. Структура содержимого этих папок одинаковая:

1. `main.cpp` – основной текст программ;
2. `linears.h` – заголовочный файл, в которых описаны классы `Vector` и `Matrix` для работы с векторами и матрицами соответственно;
3. `series.cpp`, `paral.cpp` или `split.cpp` – реализации методов этих классов.

Исходные коды программ содержатся в Приложениях 1, 2 и 3 соответственно.

Сборка всех программ осуществляется при помощи `Makefile`, см. Приложение 4.

Для программ была сгенерирована СЛАУ с 2500 переменными и $\varepsilon = 0.0001$. При таких данных последовательная программа работает 33,216 секунды, для нас это будет базовым значением. С результатами полученного ускорения можно ознакомиться в Приложении 5.

Было обнаружено, что с ростом числа процессов программа ускоряется не так, как это ожидалось. Чтобы выяснить сдерживающие факторы, было проведено профилирование программы на 2, 4, 8 и 16 процессах. С результатами профилирования можно ознакомиться в Приложении 6.

ЗАКЛЮЧЕНИЕ

1. Распараллеливание с разрезанием данных чуть-чуть эффективнее, чем с дублированием данных, но выигрых несущественнен;
2. Ускорение оказалось куда как ниже ожидаемого и стремится к $1.4\sqrt{p}$, где p – число процессов;
3. Профилирование показало, что с ростом количества процессов увеличивается время на операции Scatterv и Allreduce.

Приложение 1. *Исходный код последовательной программы*

- `series/main.cpp`

```
#include <fstream>
#include <cstdlib>
#include <iostream>
#include "linears.h"

int main(int argc, char** argv) {
        if(argc != 3) {
                std::cout << "Wrong input\n";
                return 1;
        }

        double epsilon = atof(argv[1]);
        std::ifstream in(argv[2]);
        int size;
        in >> size;
        Vector b(in, size);
        Matrix koefs(in, size);
        in.close();


        double *arrx0 = new double[size];
        for(int index = 0; index < size; index++)
                arrx0[index] = 1;
        Vector result(arrx0, size), rest = b - koefs * result, z = rest;

        double control = b.squareNorm() * epsilon * epsilon, r2 = rest.squareNorm();
        for(int iter = 0; iter < 10000 && r2 >= control; iter++) {
                Vector z_1 = koefs * z;
                double alpha = r2 / Vector::dotProduction(z_1, z);
                Vector new_rest = rest - alpha * z_1;
                double nr2 = new_rest.squareNorm(), beta = nr2 / r2;
                result = result + alpha * z;
                z = new_rest + beta * z;
                rest = new_rest;
                r2 = nr2;
        }

        result.print(std::cout);
        return 0;
}
```

- `series/linears.h`

```
#ifndef LINEARS_H
#define LINEARS_H

#include <iostream>

class Matrix {
    int size;
    double *data;
```

```
public:
    explicit Matrix(std::istream &in, int size);

    ~Matrix() { delete[] data; }

    double &operator[](int index) const;
};

class Vector {
    int size;
    double *data;

public:
    explicit Vector(int size);
    Vector(double *vec, int length): size(length), data(vec) {}
    Vector(std::istream &in, int size);
    Vector(const Vector &v);
    Vector(Vector &&v) noexcept;

    ~Vector() { delete[] data; }

    Vector &operator=(const Vector &vector);
    Vector &operator=(Vector &&vector) noexcept;

    void print(std::ostream &out) const;

    double squareNorm() const;

    double &operator[](int index) const;
    friend Vector operator+(Vector a, const Vector &b);
    friend Vector operator-(Vector a, const Vector &b);
    friend Vector operator*(const Matrix &mat, Vector vec);
    friend Vector operator*(double scalar, Vector vec);

    static double dotProduction(const Vector &a, const Vector &b);
};

#endif //LINEARS_H
```

- series/series.cpp

```
#include "linears.h"

Matrix::Matrix(std::istream &in, int length): size(length) {
        data = new double[size * size];
        for(int index = 0; index < size * size; index++) {
                in >> data[index];
                if(in.eof()) {
                        std::cerr << "reached the end of file too early\n";
                        throw std::exception();
                }
        }
}

double &Matrix::operator[](int index) const {
        return data[index];
}

Vector::Vector(int length): size(length) {
        data = new double[size];
```

```
}

Vector::Vector(std::istream &in, int length): size(length) {
        data = new double[size];
        for(int index = 0; index < size; index++) {
                in >> data[index];
                if(in.eof()) {
                        std::cerr << "reached the end of file too early\n";
                        throw std::exception();
                }
        }
}

Vector::Vector(const Vector &vector): size(vector.size) {
    data = new double[size];
    for(int index = 0; index < size; index++)
                data[index] = vector.data[index];
}

Vector::Vector(Vector &&vector) noexcept: size(vector.size) {
    data = vector.data;
    vector.data = nullptr;
}

Vector &Vector::operator=(const Vector &vector) {
        if(&vector != this) {
        for(int index = 0; index < vector.size; index++) {
                data[index] = vector.data[index];
        }
        return *this;
}

Vector &Vector::operator=(Vector &&vector) noexcept {
        if(&vector != this) {
        size = vector.size;
        delete[] data;
        data = vector.data;
        vector.data = nullptr;
        }
        return *this;
}

void Vector::print(std::ostream &out) const {
        out << data[0];
        for(int index = 1; index < size; index++)
                out << ' ' << data[index];
        out << '\n';
}

double Vector::squareNorm() const {
        return dotProduction(*this, *this);
}

double &Vector::operator[](int index) const {
        return data[index];
}

Vector operator+(Vector a, const Vector &b) {
    for(int index = 0; index < a.size; index++)
                a[index] += b[index];
    return a;
```

```
}

Vector operator-(Vector a, const Vector &b) {
    for(int index = 0; index < a.size; index++)
                a[index] -= b[index];
    return a;
}


Vector operator*(const Matrix &mat, Vector vec) {
        double *res = new double[vec.size];
        for(int row = 0; row < vec.size; row++) {
        res[row] = 0;
                for(int column = 0; column < vec.size; column++)
                        res[row] += mat[row * vec.size + column] * vec[column];
        }
        delete[] vec.data;
        vec.data = res;
        return vec;
}


Vector operator*(double scalar, Vector vec) {
        for(int index = 0; index < vec.size; index++)
                vec[index] *= scalar;
        return vec;
}

double Vector::dotProduction(const Vector &a, const Vector &b) {
        double res = 0;
        for(int index = 0; index < a.size; index++)
                res += a[index] * b[index];
        return res;
}
```

Приложение 2. *Исходный код параллельной программы с разрезанием по строкам*

- `paral/main.cpp`

```
#include <mpi.h>
#include <fstream>
#include <cstdlib>
#include <iostream>
#include "linears.h"

int rank;
int processCount;
int workZoneLeft;
int *offsets;
int *subSizes;
int subSize;
double endTime;

int main(int argc, char** argv) {
        if(argc != 3) {
                std::cout << "Wrong input\n";
                return 1;
        }
        MPI_Init(&argc, &argv);
        double startTime = MPI_Wtime();
        MPI_Comm_size(MPI_COMM_WORLD, &processCount);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        double epsilon = atof(argv[1]), *pre_sub_koefs, *pre_b;
        int N;

        if(!rank) {
                std::ifstream in(argv[2]);
                in >> N;
        MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
        workZoneLeft = N * rank / processCount;
            subSize = N * (rank + 1) / processCount - workZoneLeft;

                pre_b = new double[N];
                for(int index = 0; index < N; index++) {
                        in >> pre_b[index];
                        if(in.eof()) {
                                std::cerr << "reached the end of file too early\n";
                                throw std::exception();
                        }
                }
        MPI_Bcast(pre_b, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

        double *pre_koefs = new double[N * N];
                for(int index = 0; index < N * N; index++) {
                        in >> pre_koefs[index];
                        if(in.eof()) {
                                std::cerr << "reached the end of file too early\n";
                                throw std::exception();
                        }
                }
                in.close();
        pre_sub_koefs = new double[N * subSize];
```

```
        int *mat_offsets = new int[processCount], *mat_subSizes = new int[processCount];
        mat_offsets[0] = 0;
        for(int index = 0; index < processCount - 1; index++) {
            mat_offsets[index + 1] = (N * (index + 1) / processCount) * N;
            mat_subSizes[index] = mat_offsets[index + 1] - mat_offsets[index];
        }
        mat_subSizes[processCount - 1] = N * N - mat_offsets[processCount - 1];
        MPI_Scatterv(pre_koefs, mat_subSizes, mat_offsets, MPI_DOUBLE, pre_sub_koefs,
               subSize * N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        delete[] mat_offsets;
        delete[] mat_subSizes;
        delete[] pre_koefs;
        }
    else {
            MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
        workZoneLeft = N * rank / processCount;
            subSize = N * (rank + 1) / processCount - workZoneLeft;

        pre_b = new double[N];
        MPI_Bcast(pre_b, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

        pre_sub_koefs = new double[N * subSize];
          MPI_Scatterv(nullptr, nullptr, nullptr, MPI_DOUBLE, pre_sub_koefs, subSize * N,
                    MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }
    offsets = new int[processCount]; subSizes = new int[processCount];
    offsets[0] = 0; subSizes[0] = N / processCount;
        for(int index = 1; index < processCount; index++) {
        subSizes[index] = N * (index + 1) / processCount - N * index / processCount;
                offsets[index] = offsets[index - 1] + subSizes[index - 1];
        }

        Vector b(pre_b, N);
        Matrix koefs(pre_sub_koefs, subSize, N);

        double *arrx0 = new double[N];
        for(int index = 0; index < N; index++)
                arrx0[index] = 1;
        Vector result(arrx0, N), rest = b - koefs * result, z = rest;

        double control = b.squareNorm() * epsilon * epsilon, r2 = rest.squareNorm();
        for(int iter = 0; iter < 10000 && r2 >= control; iter++) {
                Vector z_1 = koefs * z;
                double alpha = r2 / Vector::dotProduction(z_1, z);
                Vector new_rest = rest - alpha * z_1;
                double nr2 = new_rest.squareNorm(), beta = nr2 / r2;
                result = result + alpha * z;
                z = new_rest + beta * z;
                rest = new_rest;
        r2 = nr2;
        }
        result.print(std::cout);
    if(!rank) {
        std::cerr << "It took " << endTime - startTime << " seconds\n";
    }
    delete[] offsets; delete[] subSizes;

        MPI_Finalize();
        return 0;
}
```

- paral/linears.h

```
#ifndef LINEARS_H
#define LINEARS_H

#include <mpi.h>
#include <iostream>

extern int rank;
extern int processCount;
extern int workZoneLeft;
extern int subSize;
extern int *offsets;
extern int *subSizes;
extern double endTime;

class Matrix {
    int size;
    int subSize;
    double *data;

public:
    Matrix(double *mat, int rows, int columns): size(columns), subSize(rows), data(mat) {}

    ~Matrix() { delete[] data; }

    double &operator[](int index) const;
};

class Vector {
    int size;
    double *data;

public:
    Vector(double *vec, int length): size(length), data(vec) {}
    Vector(const Vector &v);
    Vector(Vector &&v) noexcept;

    ~Vector() { delete[] data; }

    Vector &operator=(const Vector &vector);
    Vector &operator=(Vector &&vector) noexcept;

    void print(std::ostream &out) const;

    double squareNorm() const;

    double &operator[](int index) const;
    friend Vector operator+(Vector a, const Vector &b);
    friend Vector operator-(Vector a, const Vector &b);
    friend Vector operator*(const Matrix &mat, Vector vec);
    friend Vector operator*(double scalar, Vector vec);
    static double dotProduction(const Vector &a, const Vector &b);
};

#endif //LINEARS_H
```

- paral/paral.h

```
#include "linears.h"

double &Matrix::operator[](int index) const {
        return data[index];
}

Vector::Vector(const Vector &vector): size(vector.size) {
    data = new double[size];
    for(int index = 0; index < size; index++)
                data[index] = vector.data[index];
}

Vector::Vector(Vector &&vector) noexcept: size(vector.size) {
    data = vector.data;
    vector.data = nullptr;
}

Vector &Vector::operator=(const Vector &vector) {
        if(&vector != this) {
        for(int index = 0; index < vector.size; index++)
                data[index] = vector.data[index];
        }
        return *this;
}

Vector &Vector::operator=(Vector &&vector) noexcept {
        if(&vector != this) {
        size = vector.size;
        delete[] data;
        data = vector.data;
        vector.data = nullptr;
        }
        return *this;
}

void Vector::print(std::ostream &out) const {
    double *res = nullptr;
    if(!rank)
        res = new double[size];

    MPI_Gatherv(data + workZoneLeft, subSize, MPI_DOUBLE, res, subSizes, offsets,
           MPI_DOUBLE, 0, MPI_COMM_WORLD);
    endTime = MPI_Wtime();
    if(!rank) {
        out << res[0];
        for(int index = 1; index < size; index++)
                out << ' ' << res[index];
        out << '\n';
    }
    delete[] res;
}

double Vector::squareNorm() const {
        return dotProduction(*this, *this);
}

double &Vector::operator[](int index) const {
        return data[index];
}
```

```
Vector operator+(Vector a, const Vector &b) {
    for(int index = workZoneLeft; index < workZoneLeft + subSize; index++)
        a[index] += b[index];
    return a;
}

Vector operator-(Vector a, const Vector &b) {
    for(int index = workZoneLeft; index < workZoneLeft + subSize; index++)
        a[index] -= b[index];
    return a;
}


Vector operator*(const Matrix &mat, Vector vec) {
        double *res = new double[vec.size];
        MPI_Allgatherv(vec.data + workZoneLeft, subSize, MPI_DOUBLE, res, subSizes,
               offsets, MPI_DOUBLE, MPI_COMM_WORLD);
        for(int row = 0; row < subSize; row++) {
        vec[row + workZoneLeft] = 0;
                for(int column = 0; column < vec.size; column++)
                        vec[row + workZoneLeft] += mat[row * vec.size + column] *
                                res[column];
        }
    delete[] res;
        return vec;
}


Vector operator*(double scalar, Vector vec) {
    for(int index = workZoneLeft; index < workZoneLeft + subSize; index++) {
        vec[index] *= scalar;
    }
        return vec;
}

double Vector::dotProduction(const Vector &a, const Vector &b) {
        double res = 0, subRes = 0;
        for(int index = workZoneLeft; index < workZoneLeft + subSize; index++)
                subRes += a[index] * b[index];
        MPI_Allreduce(&subRes, &res, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        return res;
}
```

Приложение 3. *Исходный код параллельной программы с разрезанием матрицы и векторов*

- `split/main.cpp`

```
#include <mpi.h>
#include <fstream>
#include <cstdlib>
#include <iostream>
#include "linears.h"

int rank;
int processCount;
int *offsets;
int *subSizes;
double endTime;

int main(int argc, char** argv) {
        if(argc != 3) {
                std::cout << "Wrong input\n";
                return 1;
        }
        MPI_Init(&argc, &argv);
        double startTime = MPI_Wtime();
        MPI_Comm_size(MPI_COMM_WORLD, &processCount);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        double epsilon = atof(argv[1]);
        int N, subSize;
        double *pre_sub_b, *pre_sub_koefs;

        if(!rank) {
                std::ifstream in(argv[2]);
                in >> N;
        MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
            subSize = N * (rank + 1) / processCount - N * rank / processCount;

                double *pre_b = new double[N];
                for(int index = 0; index < N; index++) {
                        in >> pre_b[index];
                        if(in.eof()) {
                                std::cerr << "reached the end of file too early\n";
                                throw std::exception();
                        }
                }
        pre_sub_b = new double[subSize];

        offsets = new int[processCount]; subSizes = new int[processCount];
        offsets[0] = 0; subSizes[0] = N / processCount;
        for(int index = 1; index < processCount; index++) {
            subSizes[index] = N * (index + 1) / processCount - N * index / processCount;
            offsets[index] = offsets[index - 1] + subSizes[index - 1];
        }

        MPI_Scatterv(pre_b, subSizes, offsets, MPI_DOUBLE, pre_sub_b, subSize, MPI_DOUBLE,
               0, MPI_COMM_WORLD);

        double *pre_koefs = new double[N * N];
                for(int index = 0; index < N * N; index++) {
                        in >> pre_koefs[index];
```

```
                        if(in.eof()) {
                                std::cerr << "reached the end of file too early\n";
                                throw std::exception();
                        }
                }
                in.close();

        pre_sub_koefs = new double[subSize * N];
        for(int index = 0; index < N; index++) {
            MPI_Scatterv(pre_koefs + N * index, subSizes, offsets, MPI_DOUBLE,
                pre_sub_koefs + subSize * index, subSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        }
        delete[] pre_b;
        delete[] pre_koefs;
        }
    else {
        MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
        subSize = N * (rank + 1) / processCount - N * rank / processCount;

        pre_sub_b = new double[subSize];
        MPI_Scatterv(nullptr, nullptr, nullptr, MPI_DOUBLE, pre_sub_b, subSize, MPI_DOUBLE,
               0, MPI_COMM_WORLD);

        pre_sub_koefs = new double[subSize * N];
        for(int index = 0; index < N; index++) {
             MPI_Scatterv(nullptr, nullptr, nullptr, MPI_DOUBLE, pre_sub_koefs + subSize *
                    index, subSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        }
    }


        Vector b(pre_sub_b, N, subSize);
        Matrix koefs(pre_sub_koefs, N, subSize);

        double *arrx0 = new double[subSize];
        for(int index = 0; index < subSize; index++)
                arrx0[index] = 1;
        Vector result(arrx0, N, subSize), rest = b - koefs * result, z = rest;

        double control = b.squareNorm() * epsilon * epsilon, r2 = rest.squareNorm();
        for(int iter = 0; iter < 10000 && r2 >= control; iter++) {
                Vector z_1 = koefs * z;
                double alpha = r2 / Vector::dotProduction(z_1, z);
                Vector new_rest = rest - alpha * z_1;
                double nr2 = new_rest.squareNorm(), beta = nr2 / r2;
                result = result + alpha * z;
                z = new_rest + beta * z;
                rest = new_rest;
        r2 = nr2;
        }
        result.print(std::cout);
    if(!rank)
        std::cerr << "It took " << endTime - startTime << " seconds\n";
    delete[] offsets; delete[] subSizes;

        MPI_Finalize();
        return 0;
}
```

- split/linears.h

```
#ifndef LINEARS_H
#define LINEARS_H

#include <mpi.h>
#include <iostream>

extern int rank;
extern int processCount;
extern int *offsets;
extern int *subSizes;
extern double endTime;

class Matrix {
    int size;
    int subSize;
    double *data;

public:
    Matrix(double *mat, int rows, int columns): size(rows), subSize(columns), data(mat) {}

    ~Matrix() { delete[] data; }

    double &operator[](int index) const;
};

class Vector {
    int size;
    int subSize;
    double *data;

public:
    Vector(double *vec, int fullSize, int subS): size(fullSize),
       subSize(subS), data(vec) {}
    Vector(const Vector &v);
    Vector(Vector &&v) noexcept;

    ~Vector() { delete[] data; }

    Vector &operator=(const Vector &vector);
    Vector &operator=(Vector &&vector) noexcept;

    void print(std::ostream &out) const;

    double squareNorm() const;

    double &operator[](int index) const;
    friend Vector operator+(Vector a, const Vector &b);
    friend Vector operator-(Vector a, const Vector &b);
    friend Vector operator*(const Matrix &mat, Vector vec);
    friend Vector operator*(double scalar, Vector vec);

    static double dotProduction(const Vector &a, const Vector &b);
};

#endif //LINEARS_H
```

- *split/split.cpp*

```
#include "linears.h"

double &Matrix::operator[](int index) const {
        return data[index];
}

Vector::Vector(const Vector &vector): size(vector.size), subSize(vector.subSize) {
    data = new double[subSize];
    for(int index = 0; index < subSize; index++)
                data[index] = vector.data[index];
}

Vector::Vector(Vector &&vector) noexcept: size(vector.size), subSize(vector.subSize) {
    data = vector.data;
    vector.data = nullptr;
}

Vector &Vector::operator=(const Vector &vector) {
        if(&vector != this) {
        for(int index = 0; index < vector.subSize; index++)
                data[index] = vector.data[index];
        }
        return *this;
}

Vector &Vector::operator=(Vector &&vector) noexcept {
        if(&vector != this) {
        size = vector.size;
        subSize = vector.subSize;
        delete[] data;
        data = vector.data;
        vector.data = nullptr;
        }
        return *this;
}

void Vector::print(std::ostream &out) const {
    double *res = nullptr;
    if(!rank)
        res = new double[size];

        MPI_Gatherv(data, subSize, MPI_DOUBLE, res, subSizes, offsets, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
    endTime = MPI_Wtime();
    if(!rank) {
        out << res[0];
        for(int index = 1; index < size; index++)
                out << ' ' << res[index];
        out << '\n';
    }
    delete[] res;
}

double Vector::squareNorm() const {
        return dotProduction(*this, *this);
```

```
}

double &Vector::operator[](int index) const {
        return data[index];
}

Vector operator+(Vector a, const Vector &b) {
    for(int index = 0; index < a.subSize; index++)
        a[index] += b[index];
    return a;
}

Vector operator-(Vector a, const Vector &b) {
    for(int index = 0; index < a.subSize; index++)
        a[index] -= b[index];
    return a;
}


Vector operator*(const Matrix &mat, Vector vec) {
        double *subRes = new double[vec.size], *res = new double[vec.size];
        for(int row = 0; row < vec.size; row++) {
        subRes[row] = 0;
                for(int column = 0; column < vec.subSize; column++)
                        subRes[row] += mat[row * vec.subSize + column] * vec[column];
        }
    MPI_Allreduce(subRes, res, vec.size, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    for(int index = 0; index < vec.subSize; index++) {
        vec[index] = res[index + vec.size * rank / processCount];
    }
        delete[] res;
         delete[] subRes;
        return vec;
}


Vector operator*(double scalar, Vector vec) {
    for(int index = 0; index < vec.subSize; index++) {
        vec[index] *= scalar;
    }
        return vec;
}

double Vector::dotProduction(const Vector &a, const Vector &b) {
        double res = 0, subRes = 0;
        for(int index = 0; index < a.subSize; index++)
                subRes += a[index] * b[index];
        MPI_Allreduce(&subRes, &res, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
        return res;
}
```

Приложение 4. *Makefile*

Файл описания зависимостей

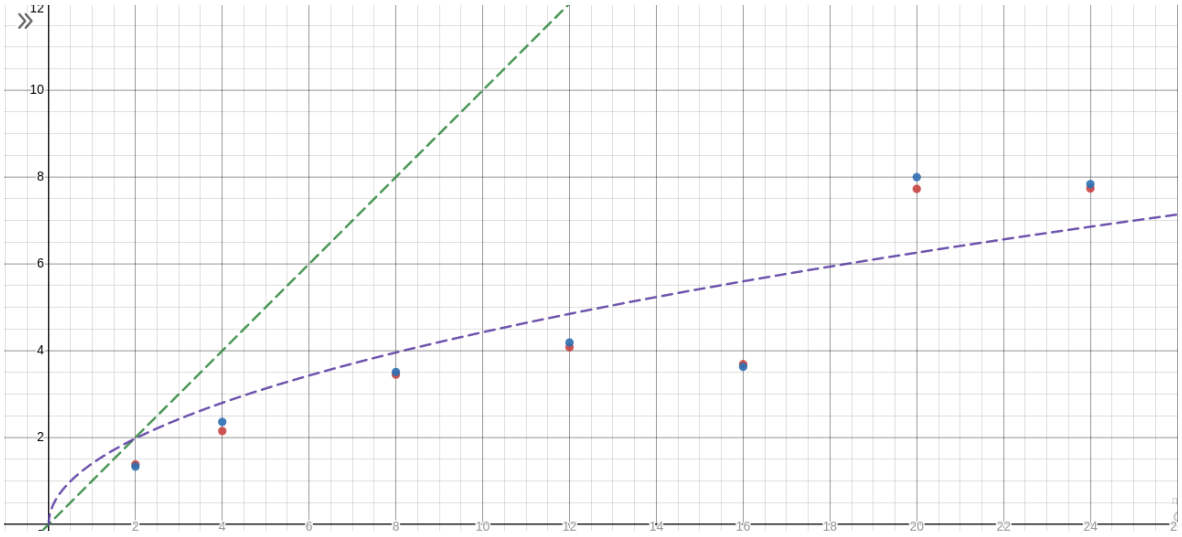
```
CC=g++
CP=mpicxx
CFLAGS=-O3 -c -Wall -std=c++14

SERIES_MAIN_FILE=series/main
SERIES_IMPLEMENTATION_FILE=series/series
SERIES_HEADER=series/linears.h
SERIES_EXE=series/series

PARAL_MAIN_FILE=paral/main
PARAL_IMPLEMENTATION_FILE=paral/paral
PARAL_HEADER=paral/linears.h
PARAL_EXE=paral/paral

PARAL_SPLIT_MAIN_FILE=split/main
PARAL_SPLIT_IMPLEMENTATION_FILE=split/split
PARAL_SPLIT_HEADER=split/linears.h
PARAL_SPLIT_EXE=split/split

SLE_GEN_SOURCE=gen_sle
SLE_GEN_EXE=gen-sle

CHECK_FILE=check

all: $(SERIES_EXE) $(PARAL_EXE) $(PARAL_SPLIT_EXE) $(SLE_GEN_EXE) $(CHECK_FILE)


$(SERIES_EXE): $(SERIES_MAIN_FILE).o $(SERIES_IMPLEMENTATION_FILE).o
	$(CC) $(SERIES_MAIN_FILE).o $(SERIES_IMPLEMENTATION_FILE).o -o $(SERIES_EXE)

$(SERIES_MAIN_FILE).o: $(SERIES_MAIN_FILE).cpp $(SERIES_HEADER)
	$(CC) $(CFLAGS) $(SERIES_MAIN_FILE).cpp -o $(SERIES_MAIN_FILE).o

$(SERIES_IMPLEMENTATION_FILE).o: $(SERIES_IMPLEMENTATION_FILE).cpp $(SERIES_HEADER)
	$(CC) $(CFLAGS) $(SERIES_IMPLEMENTATION_FILE).cpp -o $
(SERIES_IMPLEMENTATION_FILE).o


$(PARAL_EXE): $(PARAL_MAIN_FILE).o $(PARAL_IMPLEMENTATION_FILE).o
	$(CP) $(PARAL_MAIN_FILE).o $(PARAL_IMPLEMENTATION_FILE).o -o $(PARAL_EXE)

$(PARAL_MAIN_FILE).o: $(PARAL_MAIN_FILE).cpp $(PARAL_HEADER)
	$(CP) $(CFLAGS) $(PARAL_MAIN_FILE).cpp -o $(PARAL_MAIN_FILE).o

$(PARAL_IMPLEMENTATION_FILE).o: $(PARAL_IMPLEMENTATION_FILE).cpp $(PARAL_HEADER)
	$(CP) $(CFLAGS) $(PARAL_IMPLEMENTATION_FILE).cpp -o $(PARAL_IMPLEMENTATION_FILE).o


$(PARAL_SPLIT_EXE): $(PARAL_SPLIT_MAIN_FILE).o $(PARAL_SPLIT_IMPLEMENTATION_FILE).o
	$(CP) $(PARAL_SPLIT_MAIN_FILE).o $(PARAL_SPLIT_IMPLEMENTATION_FILE).o -o $
(PARAL_SPLIT_EXE)

$(PARAL_SPLIT_MAIN_FILE).o: $(PARAL_SPLIT_MAIN_FILE).cpp $(PARAL_SPLIT_HEADER)
	$(CP) $(CFLAGS) $(PARAL_SPLIT_MAIN_FILE).cpp -o $(PARAL_SPLIT_MAIN_FILE).o

$(PARAL_SPLIT_IMPLEMENTATION_FILE).o: $(PARAL_SPLIT_IMPLEMENTATION_FILE).cpp $
(PARAL_SPLIT_HEADER)
```

```
		$(CP) $(CFLAGS) $(PARAL_SPLIT_IMPLEMENTATION_FILE).cpp -o $
(PARAL_SPLIT_IMPLEMENTATION_FILE).o



$(SLE_GEN_EXE): $(SLE_GEN_SOURCE).o
		$(CC) $(SLE_GEN_SOURCE).o -o $(SLE_GEN_EXE)

$(SLE_GEN_SOURCE).o: $(SLE_GEN_SOURCE).cpp
		$(CC) $(CFLAGS) $(SLE_GEN_SOURCE).cpp -o $(SLE_GEN_SOURCE).o

$(CHECK_FILE): $(CHECK_FILE).o $(SERIES_IMPLEMENTATION_FILE).o
		$(CC) $(CHECK_FILE).o $(SERIES_IMPLEMENTATION_FILE).o -o $(CHECK_FILE)

$(CHECK_FILE).o: $(CHECK_FILE).cpp $(SERIES_HEADER)
		$(CC) $(CFLAGS) $(CHECK_FILE).cpp -o $(CHECK_FILE).o

clean:
		rm -rf */*.o
		rm -rf *.o
```

Приложение 5. *Ускорение параллельных программ по сравнению с последовательной*

Число процессов	Ускорение программы с дублированием данных, раз	Ускорение программы с разрезанием данных, раз
2	1.38	1.33
4	2.15	2.36
8	3.45	3.51
12	4.08	4.19
16	3.69	3.63
20	7.73	8.00
24	7.74	7.84

Эти же точки на одном графике:

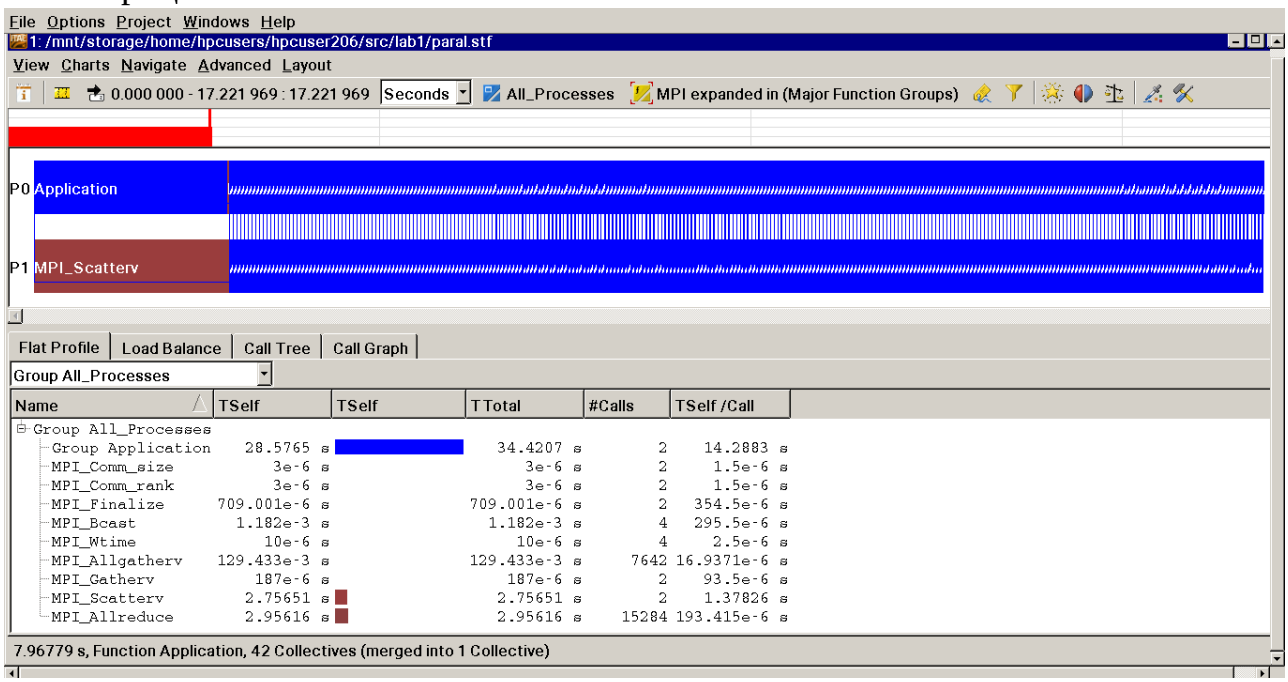


Легенда:

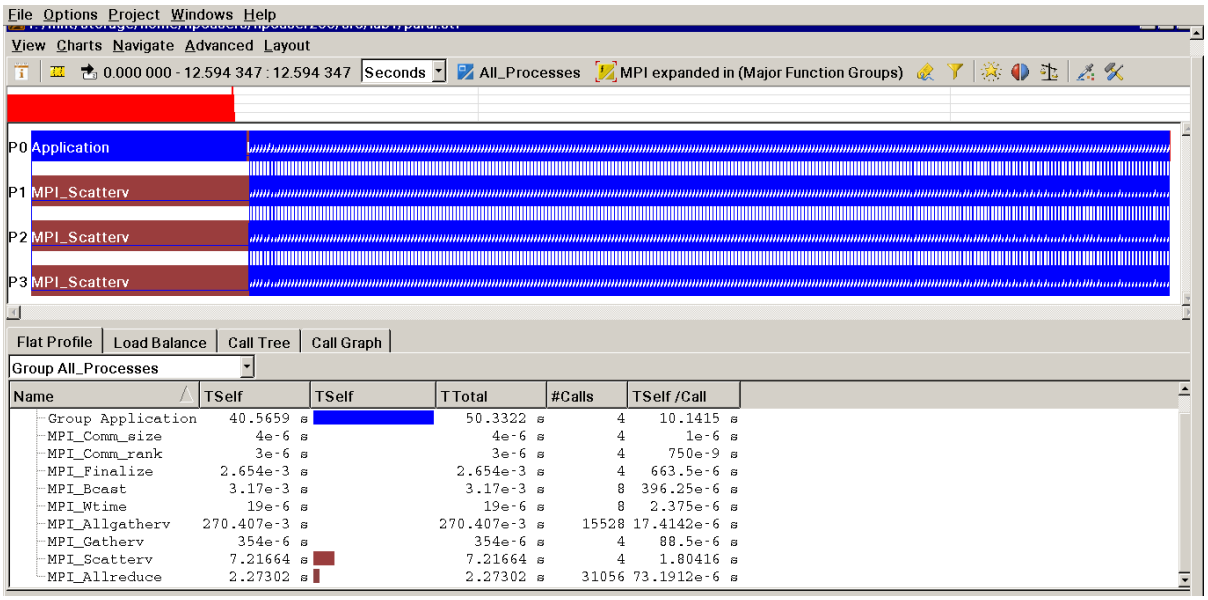
- Ox – число задействованных процессов p
- Oy – ускорение при данном числе процессов n
- Красные точки – ускорение от числа процессов первой параллельной программы
- Синие точки – ускорение от числа процессов второй параллельной программы
- Зелёная линия – график фунции $n = p$
- Фиолетовая кривая – аппроксимация функцией $n = 1.4\sqrt{p}$

Приложение 6. *Результаты профилирования программ*

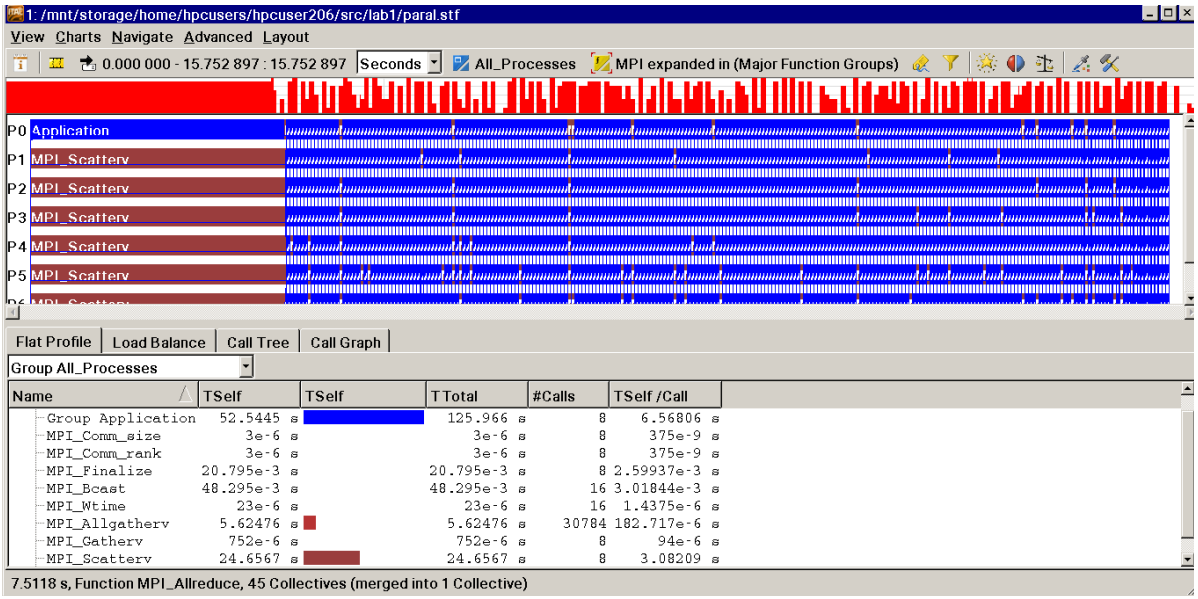
- 2 процесса



- 4 процесса



- 8 процессов


1: /mnt/storage/home/hpcusers/hpcuser206/src/lab1/paral.stf
View Charts Navigate Advanced Layout
0.000 000 - 15.752 897 : 15.752 897
Seconds
All_Processes
MPI expanded in (Major Function Groups)
P0 Application
P1 MPI_Scatterv
P2 MPI_Scatterv
P3 MPI_Scatterv
P4 MPI_Scatterv
P5 MPI_Scatterv
Flat Profile
Load Balance
Call Tree
Call Graph
Group All_Processes
Name
TSelf
TSelf
TTotal
#Calls
TSelf /Call
Group Application 52.5445 s 125.966 s 8 6.56806 s
MPI_Comm_size 3e-6 s 3e-6 s 8 375e-9 s
MPI_Comm_rank 3e-6 s 3e-6 s 8 375e-9 s
MPI_Finalize 20.795e-3 s 20.795e-3 s 8 2.59937e-3 s
MPI_Bcast 48.295e-3 s 48.295e-3 s 16 3.01844e-3 s
MPI_Wtime 23e-6 s 23e-6 s 16 1.4375e-6 s
MPI_Allgatherv 5.62476 s 5.62476 s 30784 182.717e-6 s
MPI_Gatherv 752e-6 s 752e-6 s 8 94e-6 s
MPI_Scatterv 24.6567 s 24.6567 s 8 3.08209 s
7.5118 s, Function MPI_Allreduce, 45 Collectives (merged into 1 Collective)

- 16 процессов