Федеральное государственное автономное образовательное учреждение высшего образования Санкт-Петербургский политехнический университет Петра Великого

Физико-механический институт

Высшая школа прикладной математики и вычислительной физики

Лабораторная работа №5

**Параллелизация на основе технологий MPI и OpenMP алгоритмов**

**матрично-векторного умножения для разреженных матриц**

Выполнил:

Кучиев Д.Ю.

Преподаватель:

Абрамов А.Г.

Санкт-Петербург

2025

**Цель работы**

Освоить и с помощью технологий MPI и OpenMP реализовать алгоритмы распараллеливания операции умножения разреженной матрицы большой размерности на вектор для различных применяющихся на практике способов хранения разреженных матриц.. Программные коды представлены в репозитории, а также в Приложении: https://github.com/MrDionisio/MPI\_CC

**Характеристики компьютера**

Количество ядер: 8 AMD Ryzen 6000 Series.

Объем ОЗУ: 16 Гб.

**Часть 1: Параллелизация операции умножения матрицы на вектор**

Разработать программу, параллелизующую средствами MPI и OpenMP операцию матрично-векторного умножения (для разных форматов хранения разреженных матриц). Исследовать эффективность параллелизации, сравнить эффективность параллелизации программ, разработанных с применением технологий MPI и OpenMP. Параллельный алгоритм, основанный на одномерной (ленточной) схеме разбиения матрицы на вертикальные полосы; разреженная матрица хранится в координатном формате (Coordinate Storage, COO). Координатный формат хранения означает наличие массива в котором хранятся координаты значений (индексы) и сами значения. Благодаря такому формату, мы хоть и увеличиваем массив данных в три раза, но если матрица заполнена на менее чем 33%, то выигрываем в части объема занимаемой памяти. Также такой метод позволяет уменьшить количество выполняемых операций, игнорируя нули.

Результаты работы программы (Приложение А) представлены на рисунке 1, где видна корректность выполнения исходного кода. Затем проводились аналогичные вычисления, но на матрице размерами 100 000 x 100 000 с 50 000 000 заполненными элементами. То есть было заполнено 5 % от всего размера матрицы. Время выполнения программы для различного количества процессов приведено в таблице 1, как и расчет эффективности и ускорения.

Также аналогичная программа была реализована при помощи библиотеки OpenMP и было проведено сравнение для двух подходов при распараллеливании программы.

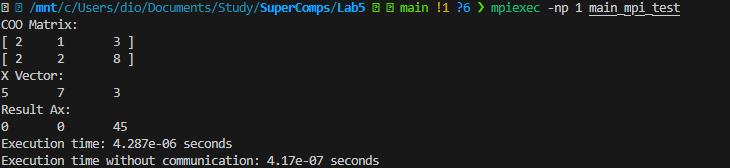


Рис. 1 Пример выполнения программы c MPI

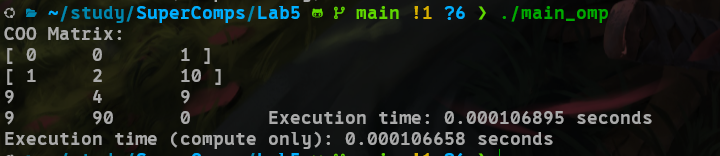


Рис. 2 Пример выполнения программы c OpenMP

Таблица 1. Расчет ускорения и эффективности работы программы.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| p | TMPI | SMPI | EMPI | TOMP | SOMP | EOMP |
| 1 | 0.89 | 1.00 | 1.00 | 1.21 | 1.00 | 1.00 |
| 2 | 0.46 | 1.93 | 0.97 | 0.77 | 1.57 | 0.79 |
| 3 | 0.51 | 1.75 | 0.58 | 0.63 | 1.92 | 0.64 |
| 4 | 0.30 | 2.97 | 0.74 | 0.46 | 2.63 | 0.66 |
| 5 | 0.29 | 3.07 | 0.61 | 0.37 | 3.27 | 0.65 |
| 6 | 0.25 | 3.56 | 0.59 | 0.31 | 3.90 | 0.65 |
| 7 | 0.21 | 4.24 | 0.61 | 0.29 | 4.17 | 0.60 |
| 8 | 0.19 | 4.68 | 0.59 | 0.27 | 4.48 | 0.56 |

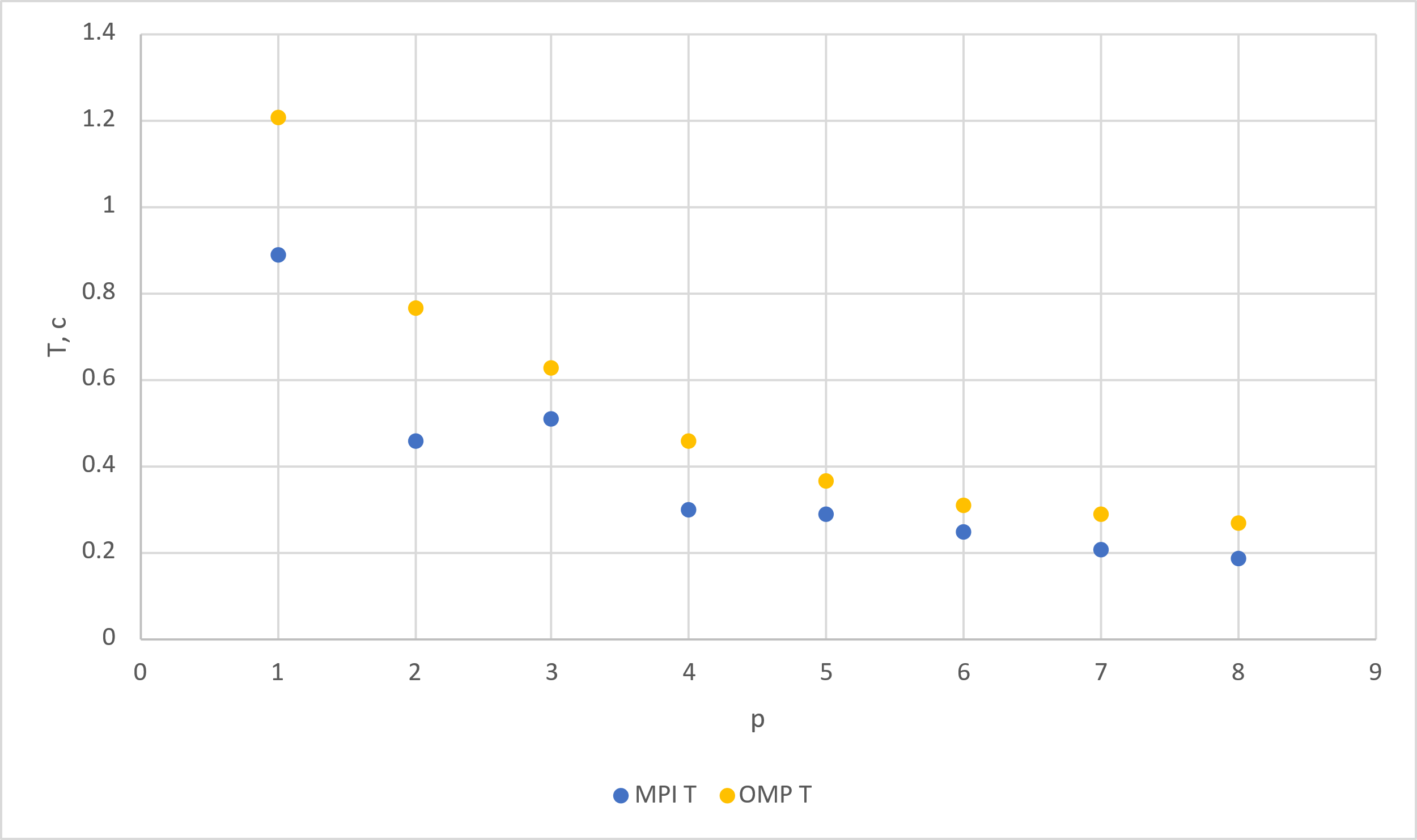


Рис. 2. Зависимость эффективности от количества процессов

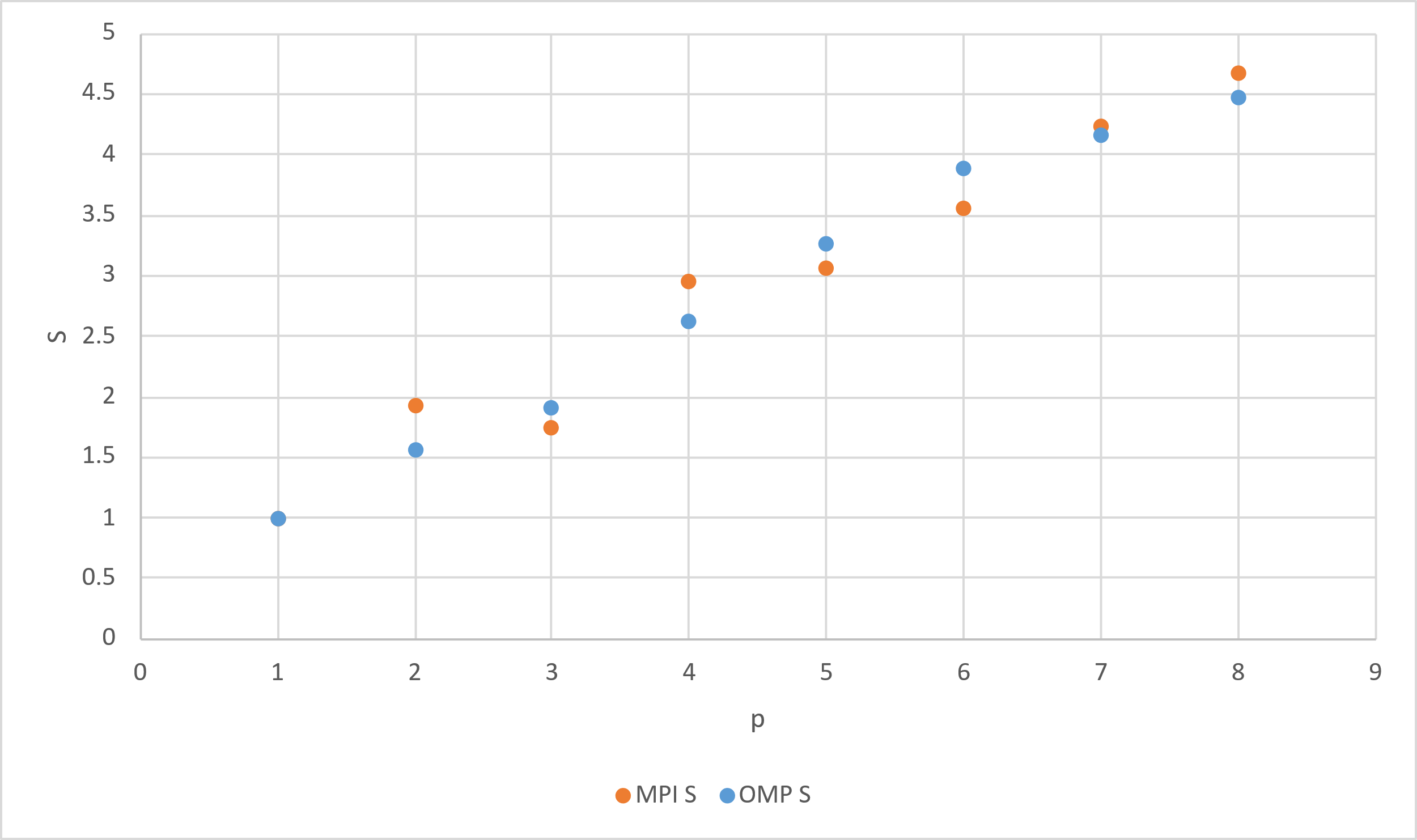


Рис. 3. Зависимость эффективности от количества процессов

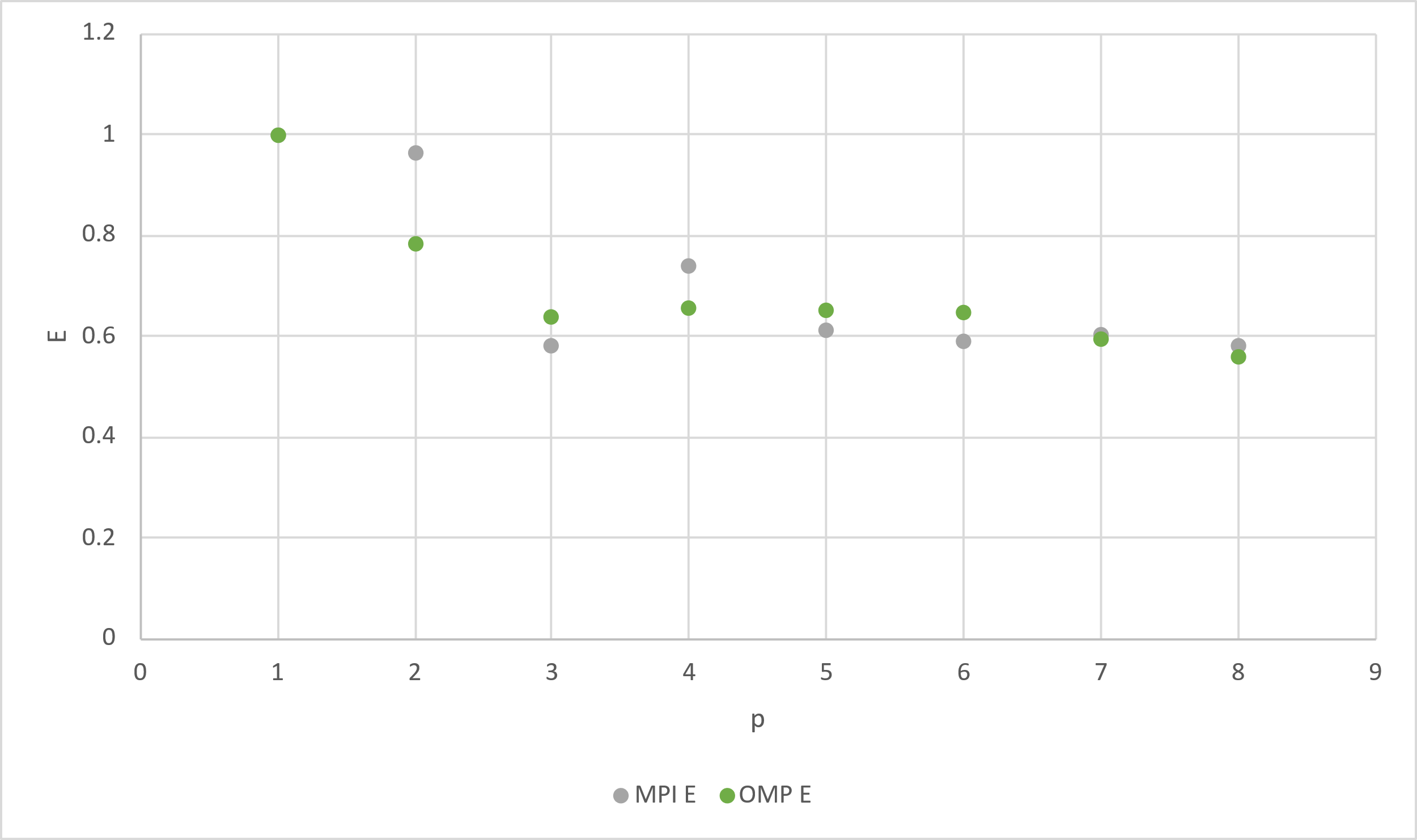


Рис. 4. Зависимость эффективности от количества процессов

Из рисунков 2-4 видно, что при увеличении процессов эффективность работы программы уменьшается. При увеличении количества процессов до 8, эффективность упала на 40% прием для обоих подходов. Также, как и ожидалось, при увеличении количества процессов, приходящихся на выполнение программы, время работы уменьшается, а ускорение растет.

**Вывод:**

В ходе выполнения лабораторной работы были реализованы и протестированы параллельные алгоритмы умножения разреженной матрицы на вектор с использованием технологий MPI и OpenMP. Для хранения разреженной матрицы применялся координатный формат (COO), позволяющий эффективно обрабатывать матрицы с низкой плотностью заполнения.

Полученные результаты подтверждают корректность реализованных алгоритмов и демонстрируют прирост производительности за счёт параллельной обработки данных. Было установлено, что с увеличением числа процессов общее время вычислений уменьшается, а ускорение возрастает. Однако эффективность при этом начинает снижаться из-за накладных расходов на синхронизацию и передачу данных между процессами.

Сравнение подходов MPI и OpenMP показало, что оба метода обеспечивают ускорение вычислений, однако MPI более устойчиво масштабируется при большом объёме данных и количестве процессов. OpenMP может быть предпочтительнее для задач с меньшим объёмом данных или при использовании одной многоядерной машины.

**Приложения**

**Приложение А. MPI реализация**

#include <mpi.h>

#include <iostream>

#include <vector>

#include <unordered\_set>

#include <cstdlib>

#include <ctime>

using namespace std;

struct COOMatrix {

    vector<size\_t> rows;

    vector<size\_t> cols;

    vector<double> values;

    size\_t nrows = 0;

    size\_t ncols = 0;

    COOMatrix() {}

    void add\_val(size\_t r\_idx, size\_t c\_idx, double val) {

        rows.push\_back(r\_idx);

        cols.push\_back(c\_idx);

        values.push\_back(val);

    }

    void random(size\_t rows\_, size\_t cols\_, size\_t nnz) {

        nrows = rows\_;

        ncols = cols\_;

        rows.clear();

        cols.clear();

        values.clear();

        std::srand(std::time(nullptr));

        std::unordered\_set<size\_t> used;

        while (rows.size() < nnz) {

            size\_t r = rand() % nrows;

            size\_t c = rand() % ncols;

            size\_t hash = r \* ncols + c;

            if (used.find(hash) == used.end()) {

                used.insert(hash);

                double val = (rand()) % 10 + 1;

                add\_val(r, c, val);

            }

        }

    }

    void print() {

        cout << "COO Matrix:\n";

        for (size\_t i = 0; i < rows.size(); i++) {

            cout << "[ " << rows[i] << "\t" << cols[i] << "\t" << values[i] << " ]\n";

        }

    }

};

int main(int argc, char\*\* argv) {

    MPI\_Init(&argc, &argv);

    int rank, size;

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

    COOMatrix A;

    vector<double> x;

    size\_t rows = 100000, cols = 100000, nnz = 50000000;

    if (rank == 0) {

        A.random(rows, cols, nnz);

        //A.print();

        x.resize(cols);

        for (auto &el : x) {

            el = (rand()) % 10 + 1;

        }

        // cout << "X Vector:\n";

        // for (auto el : x) cout << el << "\t";

        // cout << "\n";

    }

    // Замер времени: старт

    double start\_time = MPI\_Wtime();

    // Широковещание размеров

    MPI\_Bcast(&rows, 1, MPI\_UNSIGNED\_LONG, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&cols, 1, MPI\_UNSIGNED\_LONG, 0, MPI\_COMM\_WORLD);

    // Широковещание вектора x

    if (rank != 0) x.resize(cols);

    MPI\_Bcast(x.data(), cols, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    // Широковещание COO

    size\_t nnz\_total = A.values.size();

    MPI\_Bcast(&nnz\_total, 1, MPI\_UNSIGNED\_LONG, 0, MPI\_COMM\_WORLD);

    vector<size\_t> all\_rows, all\_cols;

    vector<double> all\_vals;

    if (rank == 0) {

        all\_rows = A.rows;

        all\_cols = A.cols;

        all\_vals = A.values;

    } else {

        all\_rows.resize(nnz\_total);

        all\_cols.resize(nnz\_total);

        all\_vals.resize(nnz\_total);

    }

    MPI\_Bcast(all\_rows.data(), nnz\_total, MPI\_UNSIGNED\_LONG, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(all\_cols.data(), nnz\_total, MPI\_UNSIGNED\_LONG, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(all\_vals.data(), nnz\_total, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    // Разбиение по вертикальным полосам

    size\_t stripe\_width = cols / size;

    size\_t col\_start = rank \* stripe\_width;

    size\_t col\_end = (rank == size - 1) ? cols : col\_start + stripe\_width;

    vector<size\_t> local\_idx;

    size\_t c;

    for (int i = 0; i<nnz\_total; i++) {

        c = all\_cols[i];

        if (c >= col\_start && c < col\_end) {

            local\_idx.push\_back(i);

        }

    }

    // Замер времени: старт

    double start\_time\_loc = MPI\_Wtime();

    // Локальное умножение

    vector<double> local\_result(rows, 0.0);

    for (auto idx : local\_idx) {

        size\_t r = all\_rows[idx];

        size\_t c = all\_cols[idx];

        double v = all\_vals[idx];

            local\_result[r] += v \* x[c];

    }

    // Замер времени: конец

    double end\_time\_loc = MPI\_Wtime();

    // Сбор локальных результатов

    vector<double> final\_result(rows, 0.0);

    MPI\_Reduce(local\_result.data(), final\_result.data(), rows, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

    // Замер времени: конец

    double end\_time = MPI\_Wtime();

    double elapsed = end\_time - start\_time;

    double elapsed\_loc = end\_time\_loc - start\_time\_loc;

    // Вывод результатов и времени

    if (rank == 0) {

        // cout << "Result Ax:\n";

        // for (auto val : final\_result) {

        //     cout << val << "\t";

        // }

        // cout << "\n";

        cout << "Execution time: " << elapsed << " seconds\n";

        cout << "Execution time without communication: " << elapsed\_loc << " seconds\n";

    }

    MPI\_Finalize();

    return 0;

}

**Приложение Б. OpenMP реализация**

#include <iostream>

#include <vector>

#include <unordered\_set>

#include <cstdlib>

#include <ctime>

#include <omp.h>

using namespace std;

struct COOMatrix {

    vector<size\_t> rows;

    vector<size\_t> cols;

    vector<double> values;

    size\_t nrows = 0;

    size\_t ncols = 0;

    void add\_val(size\_t r\_idx, size\_t c\_idx, double val) {

        rows.push\_back(r\_idx);

        cols.push\_back(c\_idx);

        values.push\_back(val);

    }

    void random(size\_t rows\_, size\_t cols\_, size\_t nnz) {

        nrows = rows\_;

        ncols = cols\_;

        rows.clear();

        cols.clear();

        values.clear();

        std::srand(std::time(nullptr));

        std::unordered\_set<size\_t> used;

        while (rows.size() < nnz) {

            size\_t r = rand() % nrows;

            size\_t c = rand() % ncols;

            size\_t hash = r \* ncols + c;

            if (used.find(hash) == used.end()) {

                used.insert(hash);

                double val = (rand()) % 10 + 1;

                add\_val(r, c, val);

            }

        }

    }

    void print() {

        cout << "COO Matrix:\n";

        for (size\_t i = 0; i < rows.size(); i++) {

            cout << "[ " << rows[i] << "\t" << cols[i] << "\t" << values[i] << " ]\n";

        }

    }

};

int main() {

    COOMatrix A;

    vector<double> x;

    size\_t rows = 100000, cols = 100000, nnz = 50000000;

    A.random(rows, cols, nnz);

    //A.print();

    x.resize(cols);

    for (auto &el : x) {

        el = (rand()) % 10 + 1;

    }

    // Замер общего времени

    double start\_time = omp\_get\_wtime();

    // Замер времени без инициализации

    double start\_compute\_time = omp\_get\_wtime();

    // Умножение матрицы на вектор с OpenMP

    vector<double> result(rows, 0.0);

    #pragma omp parallel for

    for (size\_t i = 0; i < nnz; ++i) {

        size\_t r = A.rows[i];

        size\_t c = A.cols[i];

        double v = A.values[i];

        #pragma omp atomic

        result[r] += v \* x[c];

    }

    double end\_compute\_time = omp\_get\_wtime();

    double end\_time = omp\_get\_wtime();

    // Вывод времени

    cout << "Execution time: " << (end\_time - start\_time) << " seconds\n";

    cout << "Execution time (compute only): " << (end\_compute\_time - start\_compute\_time) << " seconds\n";

    return 0;

}