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 №3

**Параллелизация алгоритмов вычисления определенного интеграла и типовых операций с одномерными массивами**

Выполнил:

Кучиев Д.Ю.

Преподаватель:

Абрамов А.Г.

Санкт-Петербург

2025

**Цель работы**

Освоить средства и методики распараллеливания алгоритмически относительно простых задач с помощью технологии MPI на примере задачи вычисления определенного интеграла и выполнения типовых операций с одномерными

Программные коды представлены в репозитории, а также в Приложении: https://github.com/MrDionisio/MPI\_CC

**Часть 1: *Характеристики компьютера***

Количество ядер: 8 AMD Ryzen 6000 Series.

Объем ОЗУ: 16 Гб.

**Часть 2: *Вычисление определенного интеграла***

Разработать программу, вычисляющую определенный интеграл на конечном промежутке по методу трапеций и выполнить ее распараллеливание средствами MPI. Предлагается реализовать явное распределение итераций цикла между процессами с последующим применением операции редукции для сложения рассчитанных процессами значений частичных сумм. Выполнить исследования эффективности распараллеливания при использовании технологий MPI и OpenMP.

В данной работе использовалась функция MPI:

**MPI\_Reduce (Сбор данных).** Собирает данные локальной переменной со всех процессов и позволяет провести над собранными данными операцию, в данном случае сложение, MPI\_SUM. Результат суммирования локальных переменных складывается в локальную переменную, одинаковую для всех процессов, в данном случае mpi\_sum.

Результаты работы программы:

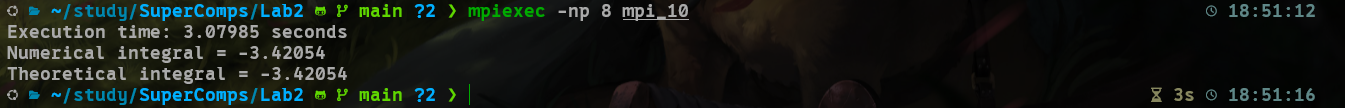


Рис. 1 Пример выполнения программы MPI



Рис. 2 Пример выполнения программы OMP

В случае с реализацией OpenMP для вычисления определенного интеграла количество кода сильно меньше, чем для реализации MPI. Необходимо меньшее количество дополнительных переменных. Код программы OpenMP представлен в Приложении А. Код программ MPI представлен в Приложении Б.

Таблица 1. Расчет ускорения и эффективности работы программы.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| p | T\_OMP | T\_MPI | S\_OMP | S\_MPI | E\_OMP | E\_MPI |
| 1 | 16.548 | 17.056 | 1.000 | 1.000 | 1.000 | 1.000 |
| 2 | 9.084 | 9.603 | 1.822 | 1.776 | 0.911 | 0.888 |
| 3 | 6.500 | 6.947 | 2.546 | 2.455 | 0.849 | 0.818 |
| 4 | 5.643 | 5.422 | 2.933 | 3.146 | 0.733 | 0.786 |
| 5 | 4.547 | 4.599 | 3.640 | 3.709 | 0.728 | 0.742 |
| 6 | 4.050 | 3.856 | 4.086 | 4.423 | 0.681 | 0.737 |
| 7 | 3.270 | 3.296 | 5.061 | 5.174 | 0.723 | 0.739 |
| 8 | 2.975 | 3.974 | 5.562 | 4.292 | 0.695 | 0.537 |

**Часть 2: Операции с одномерными массивами**

Необходимо разработать программу, реализующую одну из типовых операций с одномерными массивами (векторами) и выполнить ее распараллеливание средствами MPI и OpenMP. В MPI-программе разбиение массива и рассылка его частей по процессам должны быть реализованы с помощью коллективных коммуникационных операций стандарта.

В качестве индивидуального задания было необходимо вычислить среднее квадратичное отклонение и соответственно среднее значение.

**MPI\_Scatterv (Рассылка данных).** Разделяет данные из массива на нулевом процессе и позволяет их неравномерно распределить между всеми процессами согласно двум вспомогательным массивам, в которых указывается начальный элемент и количество отсылаемых элементов. Каждый процесс получает часть данных и переносит их в локальное хранилище.

В данном задании также использовались операции коллективного обмена MPI\_Reduce для вычисления среднего значения и среднеквадратичного отклонения. Также было увеличено количество пересчетов СКО в размере до 10000, дабы увеличить время нагрузки на процессы, чтобы минимизировать влияние временных затрат на процессы передачи данных и смоделировать большую вычислительную нагрузку.

Результаты работы программы:

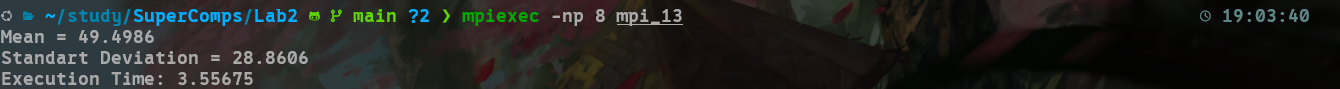


Рис. 3 Пример выполнения программы MPI



Рис. 4 Пример выполнения программы OMP

Таблица 2. Расчет ускорения и эффективности работы программы 2.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| p | T\_OMP | T\_MPI | S\_OMP | S\_MPI | E\_OMP | E\_MPI |
| 1 | 27.248 | 27.013 | 1.000 | 1.000 | 1.000 | 1.000 |
| 2 | 13.693 | 13.555 | 1.990 | 1.993 | 0.995 | 0.996 |
| 3 | 9.694 | 9.278 | 2.811 | 2.912 | 0.937 | 0.971 |
| 4 | 7.174 | 6.993 | 3.798 | 3.863 | 0.950 | 0.966 |
| 5 | 5.877 | 5.665 | 4.636 | 4.768 | 0.927 | 0.954 |
| 6 | 4.995 | 4.722 | 5.455 | 5.721 | 0.909 | 0.954 |
| 7 | 4.299 | 4.146 | 6.338 | 6.515 | 0.905 | 0.931 |
| 8 | 3.885 | 3.631 | 7.014 | 7.439 | 0.877 | 0.930 |

**Часть 3: *Исследование эффективности и ускорения для двух программ***

На рисунке 5 изображены ускорения работы программ для обеих задач за счет подключения библиотек параллелизации в зависимости от количества используемых процессоров. Видно, что в обоих случаях, ускорение растет прямо пропорционально количеству процессов.

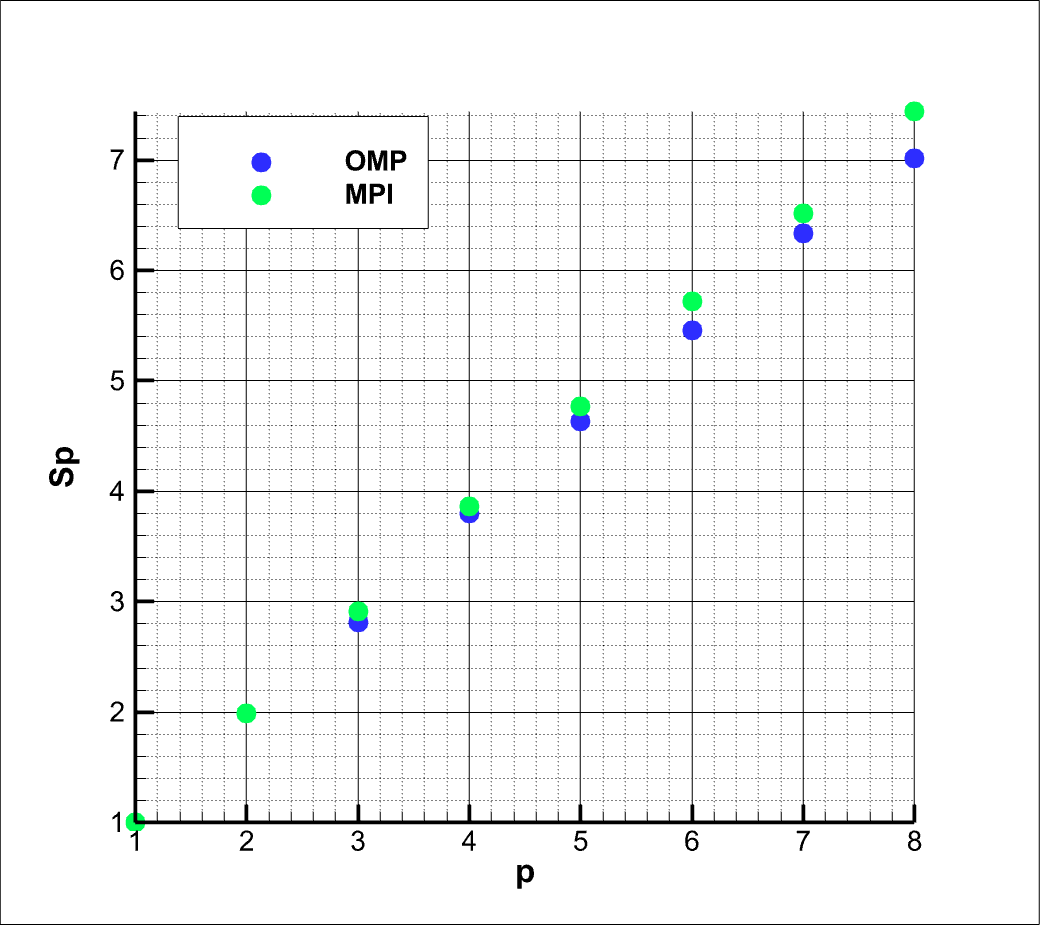
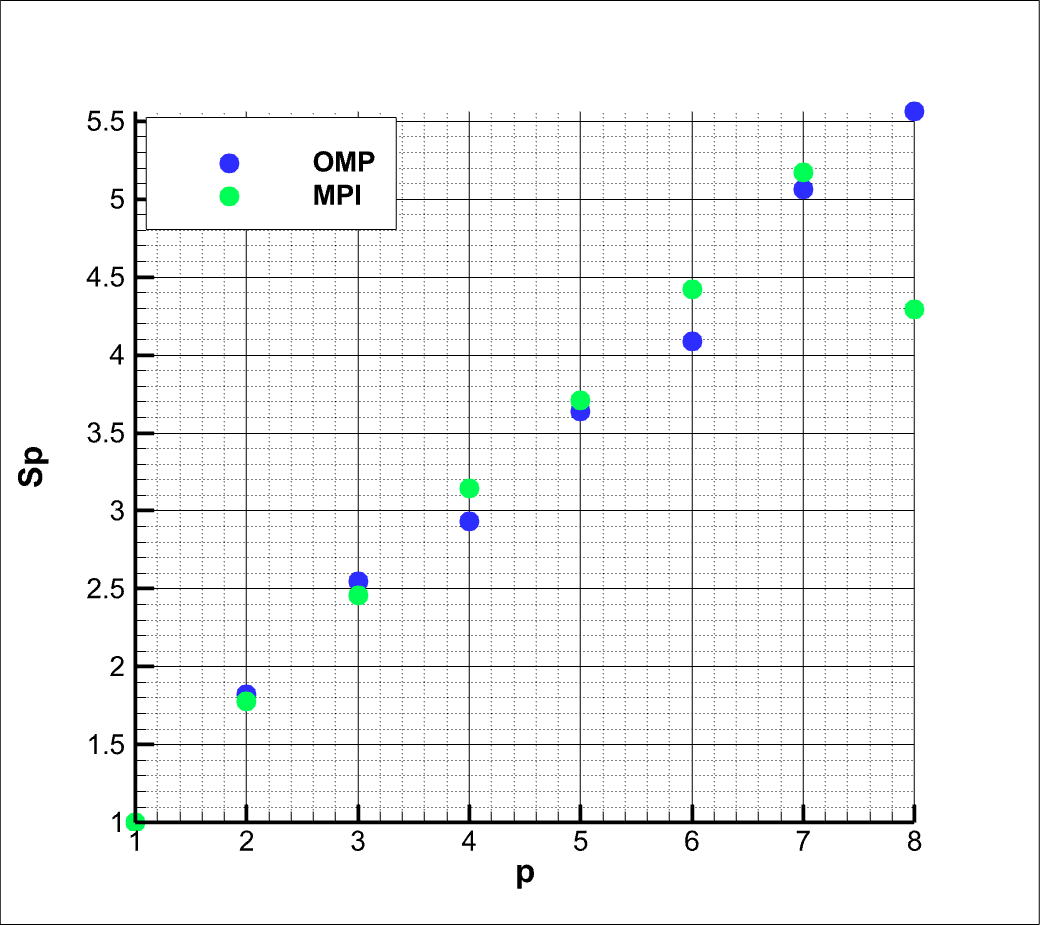
 

Рисунок 5. Зависимость ускорения от количества нитей для задачи с вычислением интеграла (слева) и с операциями над массивом (справа)

На рисунке 6 представлена зависимость эффективности от количества нитей. Наблюдается понижение эффективности при увеличении количества нитей, а также наличие некоторых выбросов и резких изменений. Также наблюдается монотонное убывание эффективности для задачи с вычислением определенного интеграла, но выход эффективности на плато в задаче связанной с операциями над массивом.

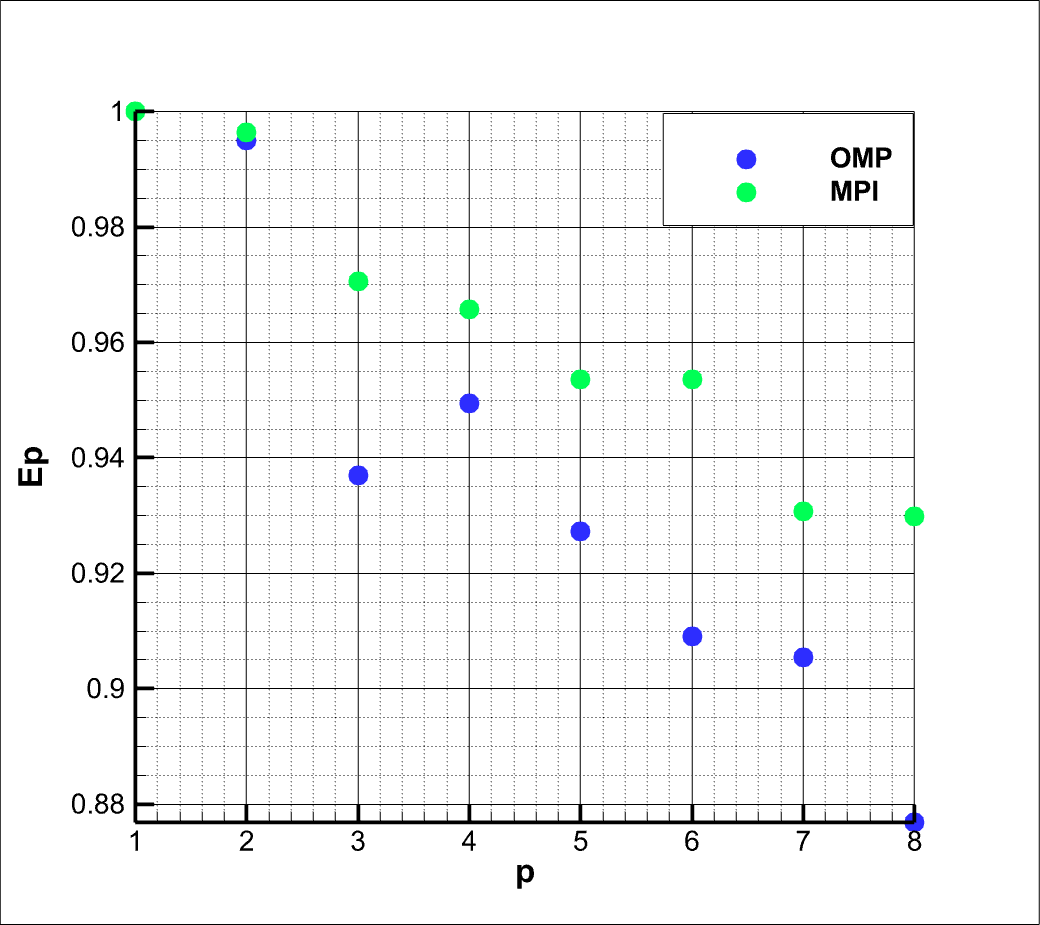
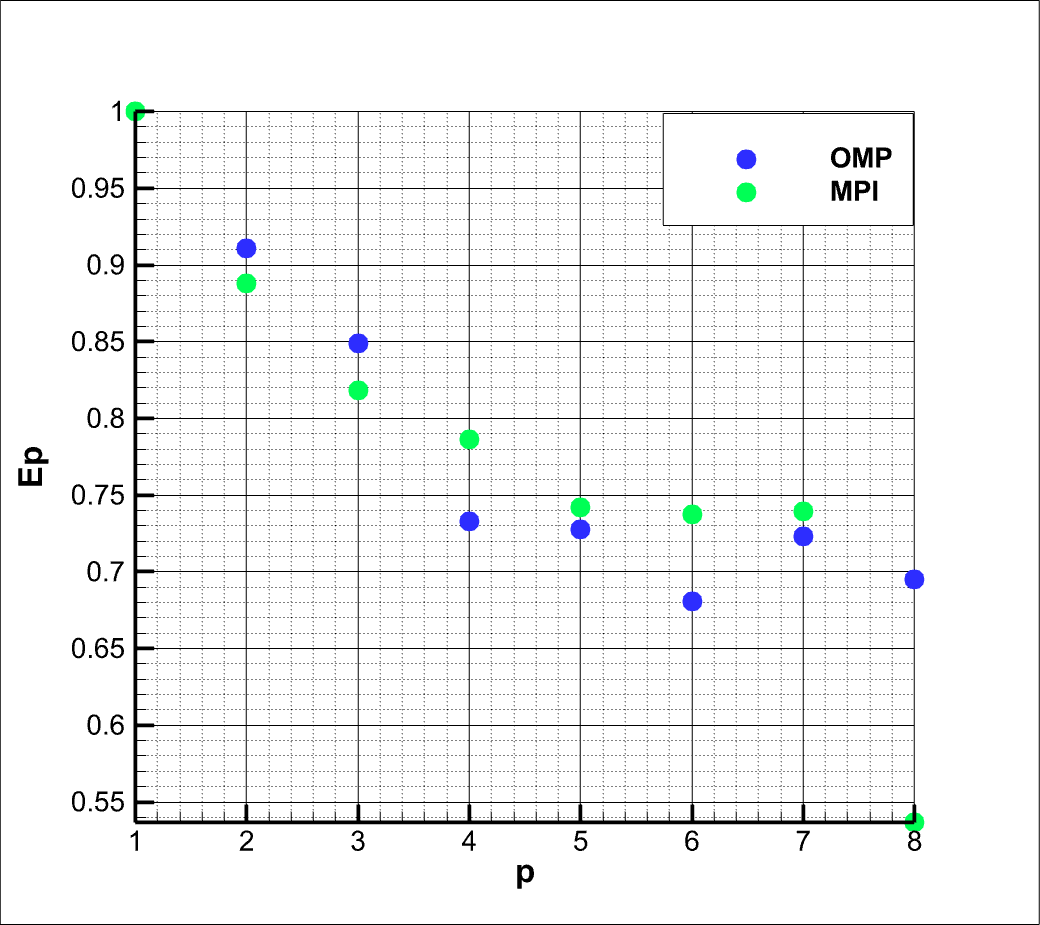
 

Рисунок 6. Зависимость эффективности от количества нитей для задачи с вычислением интеграла (слева) и с операциями над массивом (справа)

**Вывод:**

В ходе работы были реализованы параллельные алгоритмы вычисления интеграла и обработки массивов с использованием MPI и OpenMP. Исследования проводились на 8-ядерном процессоре.

1. OpenMP показал более простую реализацию при сравнимой с MPI производительности
2. Для задачи интегрирования максимальное ускорение составило 5.56× (OpenMP) и 4.29× (MPI)
3. При обработке массивов эффективность MPI была выше (0.93 против 0.88 у OpenMP)
4. Оптимальное количество потоков - 4-6, дальнейшее увеличение снижает эффективность

Полученные результаты подтверждают, что выбор технологии параллелизации должен основываться на специфике решаемой задачи и характеристиках вычислительной системы. Для задач с высокой вычислительной нагрузкой на одноузловых системах OpenMP демонстрирует лучшую производительность при меньших трудозатратах на разработку, в то время как MPI остается предпочтительным выбором для распределенных вычислений и работы с гетерогенными системами.

**Приложения**

**Приложение А. OMP Программа для вычисления определенного интеграла**

#include <iostream>

#include <cmath>

#include <chrono>

#include <omp.h>

using namespace std;

using namespace chrono;

const double pi = 4\*atan(1.0);

// Подынтегральная функция

double f(double x) {

    if(x == 0) { x += 1e-8; }

    return x \* log(sin(x));

}

int main() {

    const double a = 0;

    const double b = pi;

    const double res = -pi\*pi/2\*log(2.0);

    const int n = 1000000000;

    const double h = (b - a) / n;

    double sum = 0.0;

    auto start = high\_resolution\_clock::now();

    #pragma omp parallel for reduction(+:sum)

    for (int i = 1; i <= n; i++) {

        double x = a + (i - 0.5) \* h;

        sum += f(x);

    }

    sum \*= h;

    auto end = high\_resolution\_clock::now();

    double exec\_time = duration\_cast<microseconds>(end - start).count() / 1e6;

    cout << "Execution time: " << exec\_time << " seconds\n";

    cout << "Numerical integral = " << sum << "\n";

    cout << "Theoretical integral = " << res << "\n";

    return 0;

}

**Приложение Б. MPI Программа для вычисления определенного интеграла**

#include <iostream>

#include <math.h>

#include <mpi.h>

#include <chrono>

using namespace std;

using namespace chrono;

const double pi = 4\*atan(1.0);

double f(double x){

    if(x==0){x+=1e-8;}

    return x\*log(sin(x));

}

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

    MPI\_Init(&argc, &argv);

    double a=0;

    double b=pi;

    double res = -pi\*pi/2\*log(2.0);

    int rank, size;

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

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

    int n = 1000000000;

    int local\_n;

    double h = (b-a)/n;

    double local\_a, local\_b;

    local\_n = n/size;

    if (rank==size-1){

        local\_n = n - local\_n\*rank;

    }

    local\_a = a + rank\*local\_n\*h;

    local\_b = local\_a + h;

    //Формула Котеса

    auto start\_mpi = high\_resolution\_clock::now();

    double local\_sum = 0;

    double x;

    for (int i = 1; i < local\_n; i++) {

        x = local\_a + (i-0.5) \* h;

        local\_sum += f(x);

    }

    local\_sum \*= h;

    double mpi\_sum=0.0;

    MPI\_Reduce(&local\_sum, &mpi\_sum, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

    auto end\_mpi = high\_resolution\_clock::now();

    double mpi\_time = duration\_cast<microseconds>(end\_mpi - start\_mpi).count() / 1e6;

    if(rank==0){

        cout << "Execution time: " << mpi\_time << " seconds\n";

        cout << "Numerical integral = " << mpi\_sum << "\n";

        cout << "Theoretical integral = " << res << "\n";

    }

    MPI\_Finalize();

    return 0;

}

**Приложение В. OMP Программа для вычисления СКО**

#include <iostream>

#include <cmath>

#include <chrono>

#include <random>

#include <omp.h>

using namespace std;

using namespace chrono;

int main() {

    const int n = 1e6;

    const int iterations = 10000;

    // Инициализация данных

    double\* x = new double[n];

    srand(1);

    for(int i = 0; i < n; i++) {

        x[i] = static\_cast<double>(rand() % 100);

    }

    // Вычисление среднего значения

    double sum = 0.0;

    #pragma omp parallel for reduction(+:sum)

    for(int i = 0; i < n; i++) {

        sum += x[i];

    }

    double mean = sum / n;

    cout << "Mean = " << mean << endl;

    // Вычисление стандартного отклонения

    int i;

    auto start = high\_resolution\_clock::now();

    double variance = 0.0;

    for(int k=0; k<10000;k++){

        variance = 0.0;

        #pragma omp parallel for reduction(+:variance)

        for(i = 0; i < n; i++) {

            variance += (x[i] - mean) \* (x[i] - mean);

        }

    }

    double std\_dev = sqrt(variance / n);

    cout << "Standard Deviation = " << std\_dev << endl;

    auto end = high\_resolution\_clock::now();

    double exec\_time = duration\_cast<microseconds>(end - start).count() / 1e6;

    cout << "Execution Time: " << exec\_time << " seconds" << endl;

    // Очистка памяти

    delete[] x;

    return 0;

}

**Приложение Г. MPI Программа для вычисления СКО**

#include <iostream>

#include <math.h>

#include <mpi.h>

#include <chrono>

#include <random>

using namespace std;

using namespace chrono;

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);

    int n = 1e6;

    double \*x = nullptr;

    int \*sendcount{nullptr},  \*displace{nullptr};

    int local\_n = n/size;

    if (rank==0){

        srand(1);

        x = new double[n];

        for(int i=0; i<n; i++){

            x[i]=static\_cast<double>(rand()%100);

        }

        sendcount = new int[size];

        displace = new int[size];

        for(int i=0; i<size; i++){

            sendcount[i] = local\_n;

            if(i==size-1){

                sendcount[size-1]=n - local\_n \* (size-1);

            }

        }

        displace[0] = 0;

        for(int i=1; i<size; i++){

            displace[i] = displace[i-1]+sendcount[i-1];

        }

    }

    if(rank==size-1){

        local\_n = n - local\_n \* rank;

    }

    double\* local\_x = new double[local\_n];

    MPI\_Scatterv(x, sendcount, displace, MPI\_DOUBLE, local\_x, local\_n, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    double local\_mean=0.0;

    for(int i=0; i<local\_n; i++){

        local\_mean+=local\_x[i];

    }

    double mean;

    MPI\_Reduce(&local\_mean, &mean, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

    if(rank==0){

        mean/=n;

        cout <<"Mean = "<< mean<<endl;

    }

    MPI\_Bcast(&mean, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    auto start\_mpi = high\_resolution\_clock::now();

    double std\_dev\_local;

    for(int k=0; k<10000; k++){

        std\_dev\_local=0.0;

        for(int i=0; i<local\_n; i++){

            std\_dev\_local += (local\_x[i]-mean)\*(local\_x[i]-mean);

        }

    }

    double std\_dev;

    MPI\_Reduce(&std\_dev\_local, &std\_dev, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

    if(rank==0){

        std\_dev = sqrt(std\_dev/n);

        cout <<"Standart Deviation = "<< std\_dev <<endl;

    }

    auto end\_mpi = high\_resolution\_clock::now();

    double mpi\_time = duration\_cast<microseconds>(end\_mpi - start\_mpi).count() / 1e6;

    if(rank==0){

        cout << "Execution Time: " << mpi\_time << endl;

        delete[] x, sendcount, displace;

    }

    delete[] local\_x;

    MPI\_Finalize();

    return 0;

}