###### 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 ЛАБОРАТОРНОЙ РАБОТЫ

«Параллельная реализация решения системы линейных алгебраических уравнений с помощью MPI»

студента 2 курса, 20204 группы

**Дронова Дениса Юрьевича**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Кандидат технических наук

А. Ю. Власенко

Новосибирск 2022

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc98584822)

[ЗАДАНИЕ 3](#_Toc98584823)

[ХОД РАБОТЫ 4](#_Toc98584824)

[ЗАКЛЮЧЕНИЕ 5](#_Toc98584825)

[Приложение 1. *Графики* 6](#_Toc98584826)

[Приложение 2. *Профилирование MPI\_1* 7](#_Toc98584827)

[Приложение 3. *Профилирование MPI\_2* 9](#_Toc98584828)

[Приложение 4. *Последовательная программа* 11](#_Toc98584829)

[Приложение 5. *Параллельная программа 1* 15](#_Toc98584830)

[Приложение 6. *Параллельная программа 2* 21](#_Toc98584831)

# ЦЕЛЬ

Ознакомиться с основными функциями MPI. Реализовать программу решения системы линейных алгебраических уравнений, используя функционал MPI. Оценить эффективность и ускорение программ при различном числе процессов, предоставленных для решения задачи.

# ЗАДАНИЕ

1. Написать 3 программы (одну последовательную и 2 параллельные) на языке C или C++, которые реализуют итерационный алгоритм решения системы линейных алгебраических уравнений вида Ax=b в соответствии с выбранным вариантом. Здесь A– матрица размером N×N, x и b – векторы длины N. Тип элементов –double.
2. Параллельные программы реализовать с помощью MPI с разрезанием матрицы A по строкам (в первой программе) или по столбцам (во второй) на близкие по размеру, возможно не одинаковые, части.   
   Программа 1 (MPI\_1): векторы x и b дублируются в каждом MPI-процессе,  
   Программа 2 (MPI\_2): векторы x и b разрезаются между MPI-процессами.  
   Уделить внимание тому, чтобы при запуске программы на различном  
   числе MPI-процессов решалась одна и та же задача (исходные данные  
   заполнялись одинаковым образом).
3. Замерить время работы последовательного варианта программы и 2-х параллельных при использовании различного числа процессорных ядер. Минимально на 2, 4, 8, 16, 24 (на каждом из наших узлов кластера по 12 ядер), но чем на большем количестве процессов будут выполнены замеры, тем лучше. Также чем больше замеров будет выполнено на одном и том же количестве процессов, тем лучше. В этом случае для построения графиков следует брать минимальное время.  
   Построить графики зависимости времени работы программы, ускорения и эффективности распараллеливания от числа используемых ядер. Исходные данные, параметры N и ε подобрать таким образом, чтобы решение задачи на одном ядре занимало не менее 30 секунд.
4. Выполнить профилирование двух вариантов программы с помощью jumpshot или ITAC (Intel Trace Analyzer and Collecter) при использовании 16-и ядер.
5. На основании полученных результатов сделать вывод о целесообразности использования одного или второго варианта программ

# ХОД РАБОТЫ

В первую очередь был реализован последовательный алгоритм решения СЛАУ методом сопряженных градиентов. Затем на основе последовательной программы были написаны два варианта MPI-программ, реализующих разные подходы к разрезанию данных между процессами, описанные в задании. В последующем все варианты алгоритмов были подвергнуты тестированию на корректность решения при разном числе процессов и разных размерах исходной матрицы (параметр N). Также были проведены замеры времени исполнения каждой программы при варьировании числа процессов и при одном размере матрицы для всех вариантов. Предварительно был подобран параметр N (размер матрицы) так, чтобы время исполнения последовательной программы не было меньше 30 секунд. На основании полученных данных были построены таблицы и графики времени исполнения, эффективности и ускорения в зависимости от числа процессов при одинаковом параметре N (см. Приложение). В заключение было проведено профилирование двух вариантов параллельных программ при использовании 16-и ядер.

Подобранный параметр N=1400

|  |  |  |  |
| --- | --- | --- | --- |
| ***MPI\_1*** | | | |
| **Число процессов** | **Время выполнения** | **Эффективность** | **Ускорение** |
| 1 | 50,55771 | 100% | 1 |
| 2 | 24,25912 | 104% | 2,084069981 |
| 4 | 13,17984 | 96% | 3,835986981 |
| 8 | 6,86043 | 92% | 7,369467711 |
| 12 | 4,65815 | 90% | 10,85360749 |
| 16 | 3,71408 | 85% | 13,6124524 |
| 24 | 2,66101 | 79% | 18,9994224 |

|  |  |  |  |
| --- | --- | --- | --- |
| ***MPI\_2*** | | | |
| **Число процессов** | **Время выполнения** | **Эффективность** | **Ускорение** |
| 1 | 50,870756 | 100% | 1 |
| 2 | 24,62584 | 103% | 2,065747036 |
| 4 | 13,263544 | 96% | 3,835381856 |
| 8 | 6,743541 | 94% | 7,543626709 |
| 12 | 4,689834 | 90% | 10,84702699 |
| 16 | 3,804276 | 84% | 13,37199404 |
| 24 | 2,935699 | 72% | 17,32832828 |

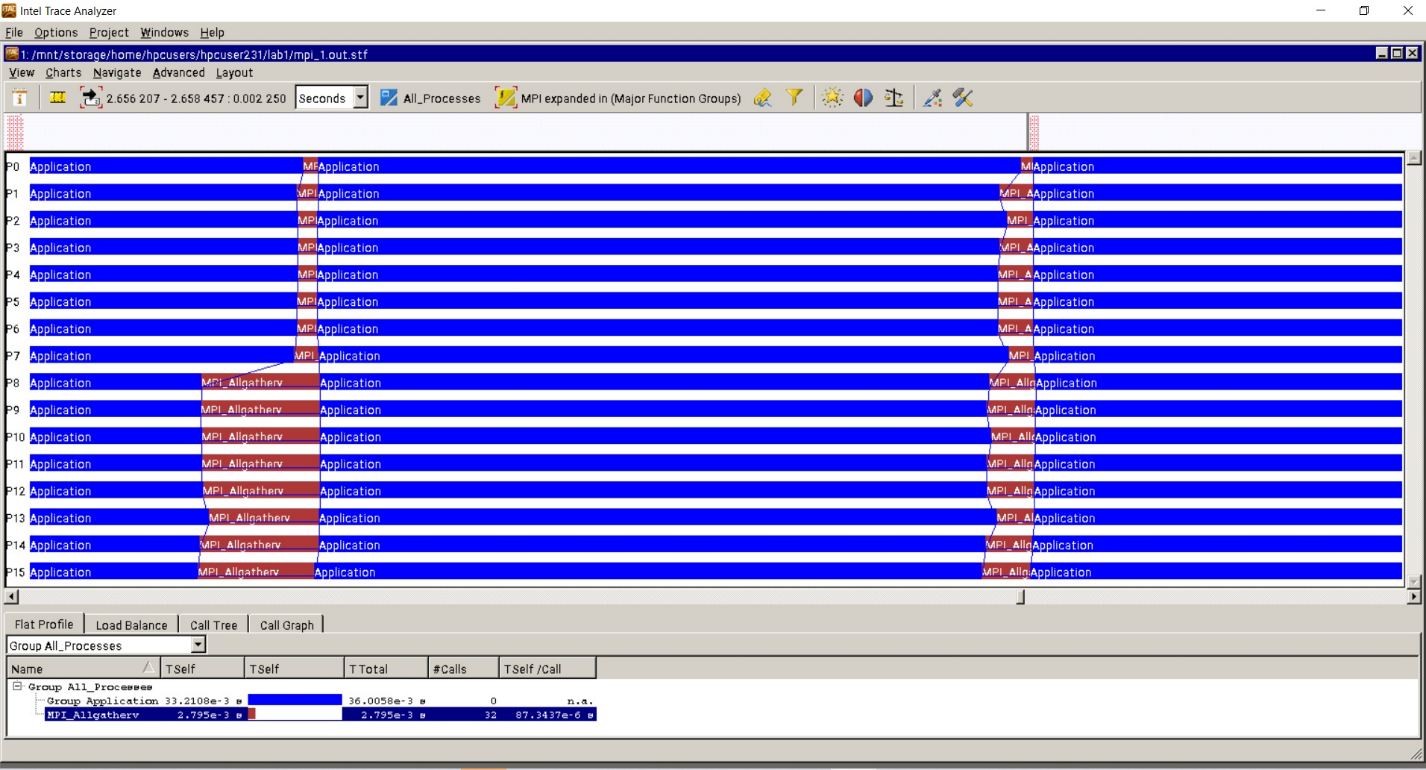
# ЗАКЛЮЧЕНИЕ

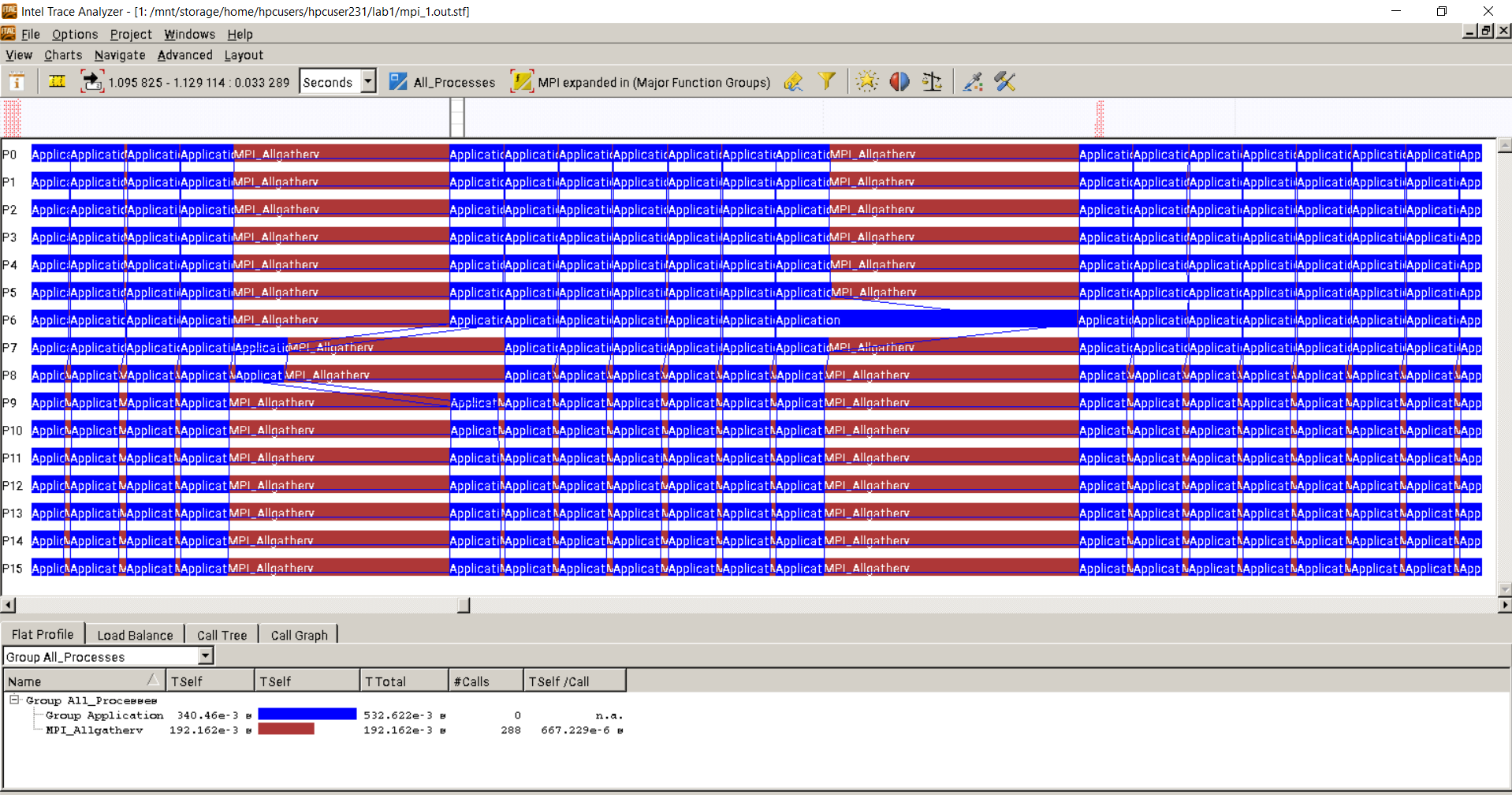
После анализа данных, полученных в результате замеров времени исполнения программ, в которых используются MPI-преобразования, можно сделать вывод о том, что использование функций библиотеки MPI и исполнения программы на большом количестве процессов может оказаться довольно эффективным методом ускорения исходной программы. Однако с увеличением числа процессов/ядер увеличиваются накладные расходы на передачу информации между ними, что может привести к увеличению времени исполнения программ, т. к. зачастую процессы располагаются не на одном узле, а на нескольких, соединенных линиями коммуникации. После анализа профилирования программ становится понятно, что умножение векторов на скаляр и скалярное произведение векторов выполняются гораздо быстрее по сравнению с умножением матрицы на вектор, поэтому распараллеливание первых двух становится нецелесообразным ввиду потенциально лишнего времени, затрачиваемого на распределение данных между процессами, которое может быть не компенсировано более быстрым умножением меньших кусков векторов. Значительную долю времени среди MPI-функций занимают MPI\_Scatter и MPI\_Reduce в функции умножения матрицы на вектор.

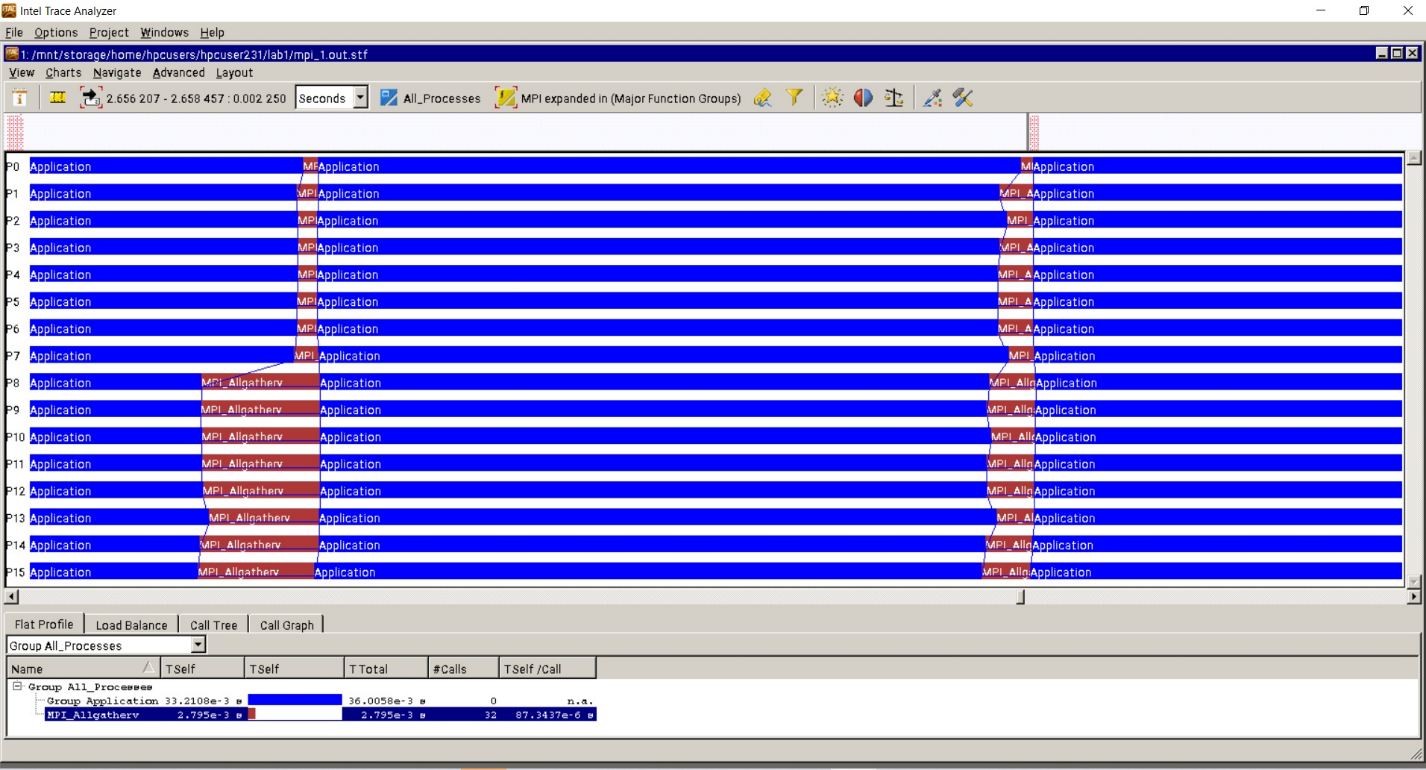
# Приложение 1. *Графики*

# Приложение 2. *Профилирование MPI\_1*

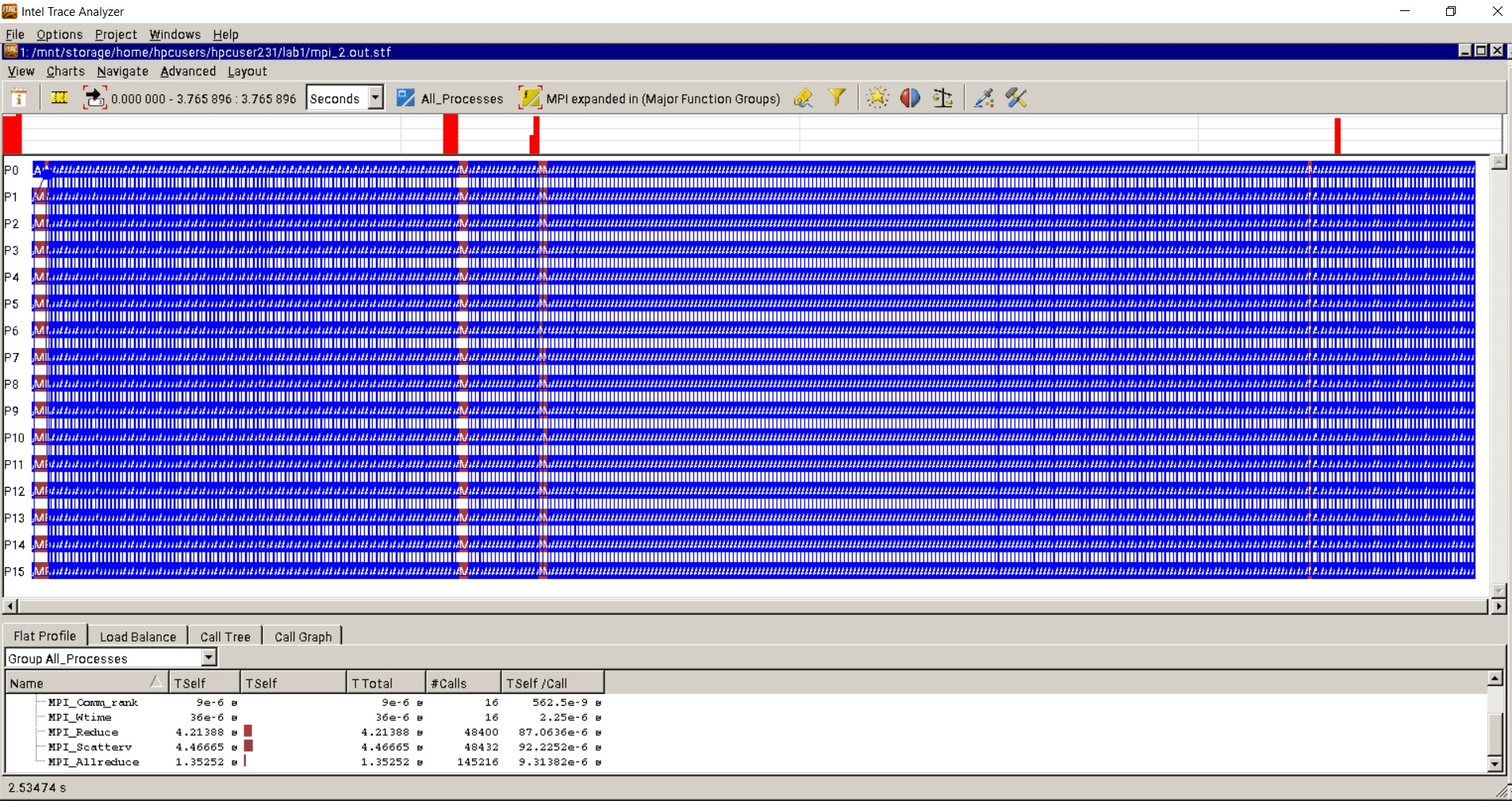


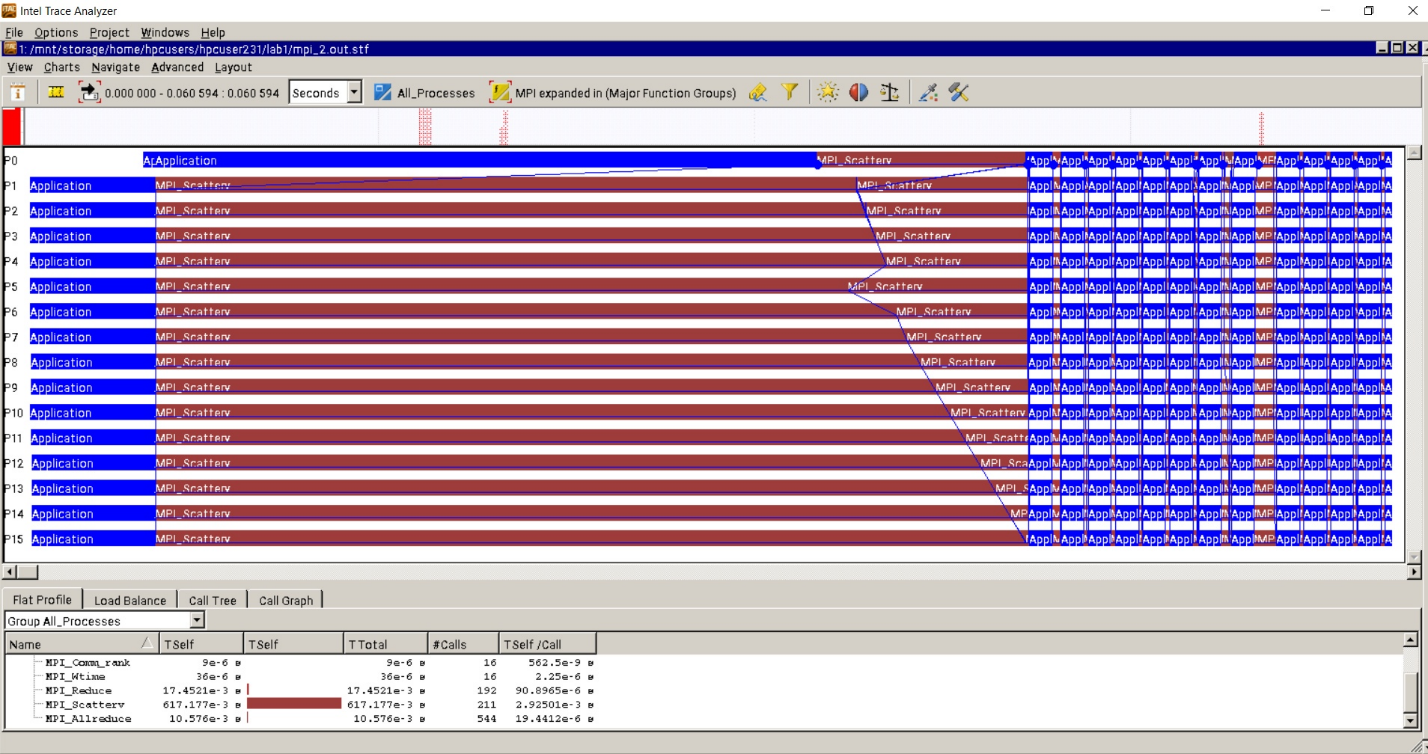


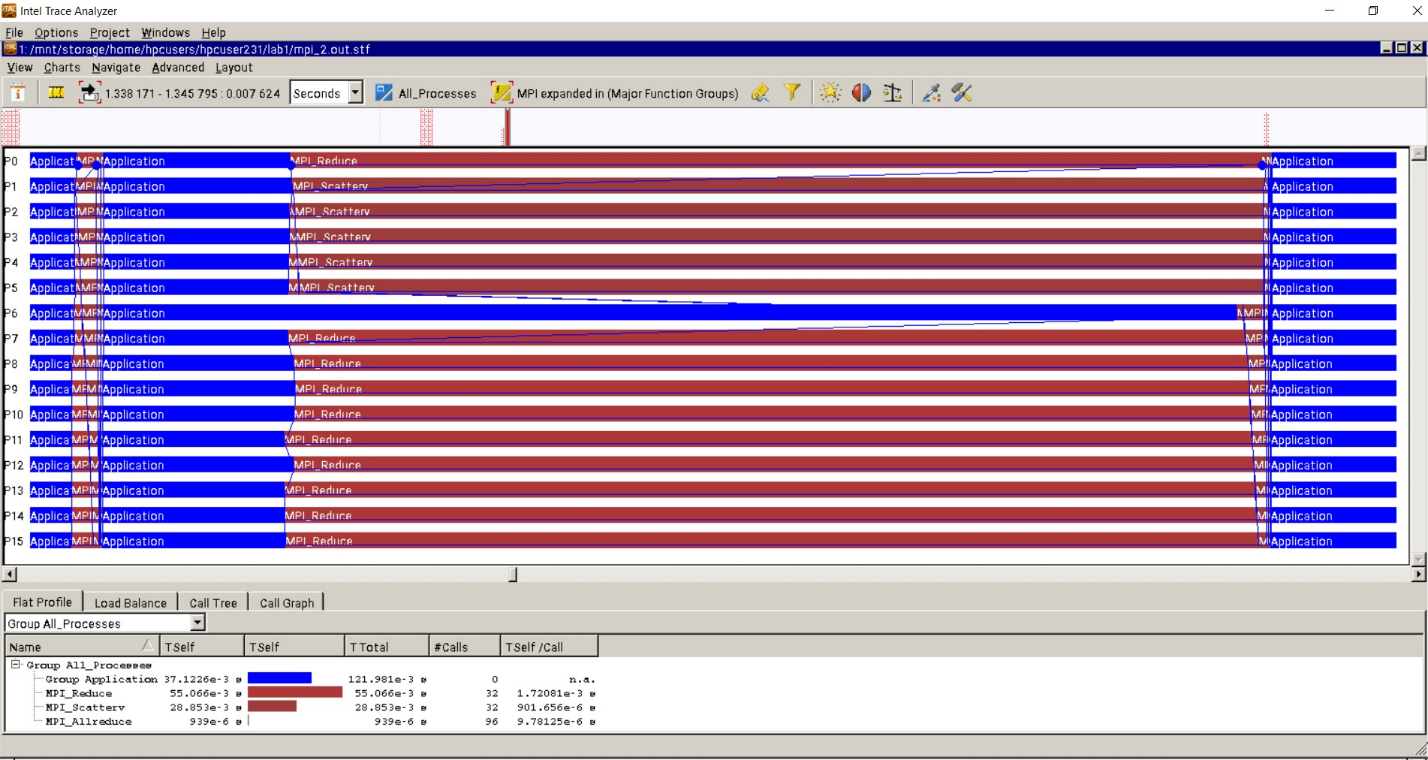




# Приложение 3. *Профилирование MPI\_2*







# Приложение 4. *Последовательная программа*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <ctime>

#define N 1400

#define epsilon 10e-5

double sqrNormOfVector(double \*vector)

{

    double result = 0;

    for (int i = 0; i < N; i++)

    {

        result += vector[i] \* vector[i];

    }

    return result;

}

double \*MatrixMultVector(double \*A, double \*vector)

{

    double \*result = (double \*)calloc(N, sizeof(double));

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            result[i] += A[i \* N + j] \* vector[j];

        }

    }

    return result;

}

double \*SubVectors(double \*vect1, double \*vect2)

{

    double \*result = (double \*)malloc(N \* sizeof(double));

    for (int i = 0; i < N; i++)

    {

        result[i] = vect1[i] - vect2[i];

    }

    return result;

}

double \*SumVectors(double \*vect1, double \*vect2)

{

    double \*result = (double \*)malloc(N \* sizeof(double));

    for (int i = 0; i < N; i++)

    {

        result[i] = vect1[i] + vect2[i];

    }

    return result;

}

double VectorsMult(double \*vect1, double \*vect2)

{

    double result = 0;

    for (int i = 0; i < N; i++)

    {

        result += vect1[i] \* vect2[i];

    }

    return result;

}

double \*ScalarVectMult(double \*vector, double scalar)

{

    double \*result = (double \*)malloc(N \* sizeof(double));

    for (int i = 0; i < N; i++)

    {

        result[i] = vector[i] \* scalar;

    }

    return result;

}

int main()

{

    srand(time(0));

    double \*A = (double \*)malloc(N \* N \* sizeof(double));

    double \*b = (double \*)malloc(N \* sizeof(double));

    double \*r = (double \*)malloc(N \* sizeof(double));

    double \*prev\_r = (double \*)malloc(N \* sizeof(double));

    double \*z = (double \*)malloc(N \* sizeof(double));

    double \*X = (double \*)calloc(N, sizeof(double));

    // INITIALIZING OF VARIABLES

    for (int i = 0; i < N; i++)

    {

        b[i] = (double)(rand() % 100);

        r[i] = b[i]; // r0 = b - Ax0, x0 - нулеой вектор

        z[i] = r[i]; // z0 = r0

        X[i] = 0;

        for (int j = i; j < N; j++)

        {

            if (i == j)

            {

                A[i \* N + j] = (double)(rand() % 100) + 200;

            }

            else

            {

                A[i \* N + j] = A[j \* N + i] = (double)(rand() % 100);

            }

        }

    }

    double start = clock();

    double \*temp = NULL;

    double alpha = 0;

    double beta = 0;

    int cntr = 0;

    int cnt = 0;

    double square\_b\_len = 0; // square of vector b length

    for (int i = 0; i < N; i++)

    {

        square\_b\_len += b[i] \* b[i];

    }

    double restrictor = epsilon \* epsilon \* square\_b\_len;

    while (cntr < 3)

    {

        temp = MatrixMultVector(A, z); // A\*z(n)

        double oldsqrR = VectorsMult(r, r);

        alpha = oldsqrR / VectorsMult(temp, z); // alpha(n+1) = (r(n), r(n)) / (Az(n), z(n))

        X = SumVectors(X, ScalarVectMult(z, alpha)); // X(n+1) = X(n) + alpha(n+1)\*z(n)

        prev\_r = r;

        r = SubVectors(r, ScalarVectMult(temp, alpha)); // r(n+1) = r(n) - alpha(n+1)Az(n)

        beta = VectorsMult(r, r) / oldsqrR; // beta(n+1) = (r(n+1), r(n+1)) / (r(n), r(n))

        z = SumVectors(r, ScalarVectMult(z, beta)); // z(n+1) = r(n+1) + beta(n+1)z(n)

        if (sqrNormOfVector(r) < restrictor)

        {

            cntr++;

        }

        cnt++;

    }

    double total\_time = (clock() - start) / CLOCKS\_PER\_SEC;

    printf("Iterations count: %d \n", cnt);

    printf("Total time: %f \n", total\_time);

    free(A);

    free(b);

    free(r);

    free(prev\_r);

    free(z);

    free(X);

    return 0;

}

# Приложение 5. *Параллельная программа 1*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include "mpi.h"

#define N 1400

#define epsilon 10e-5

void MatrixMultVector(double \*A, double \*vector, double \*result, int height, int width)

{

    for (int i = 0; i < height; i++)

    {

        for (int j = 0; j < width; j++)

        {

            result[i] += A[i \* width + j] \* vector[j];

        }

    }

}

void parallelMatrixMultVector(double \*A\_part, double \*vector, double \*result,

                              int A\_p\_width, int A\_p\_height, int procSize, int procRank)

{

    // A\_part is N \* rowsNum[procRank]

    // res is vector (one column matrix) of A\_part height

    double \*tmp = (double \*)calloc(A\_p\_height, sizeof(double));

    MatrixMultVector(A\_part, vector, tmp, A\_p\_height, A\_p\_width);

    // array of numbers of elemets send by each process

    int \*recvcounts = (int \*)calloc(procSize, sizeof(int));

    for (int i = 0; i < procSize; ++i)

    {

        recvcounts[i] = A\_p\_width / procSize;

        recvcounts[i] += (i < A\_p\_width % procSize);

    }

    int \*displs = (int \*)calloc(procSize, sizeof(int));

    for (int i = 1; i < procSize; ++i)

    {

        displs[i] = displs[i - 1] + recvcounts[i - 1];

    }

    MPI\_Allgatherv(tmp, A\_p\_height, MPI\_DOUBLE, result, recvcounts, displs, MPI\_DOUBLE, MPI\_COMM\_WORLD);

    free(tmp);

    free(displs);

    free(recvcounts);

}

double sqrNormOfVector(double \*vector)

{

    double result = 0;

    for (int i = 0; i < N; i++)

    {

        result += vector[i] \* vector[i];

    }

    return result;

}

double \*SubVectors(double \*vect1, double \*vect2)

{

    double \*result = (double \*)malloc(N \* sizeof(double));

    for (int i = 0; i < N; i++)

    {

        result[i] = vect1[i] - vect2[i];

    }

    return result;

}

double \*SumVectors(double \*vect1, double \*vect2)

{

    double \*result = (double \*)malloc(N \* sizeof(double));

    for (int i = 0; i < N; i++)

    {

        result[i] = vect1[i] + vect2[i];

    }

    return result;

}

double VectorsMult(double \*vect1, double \*vect2)

{

    double result = 0;

    for (int i = 0; i < N; i++)

    {

        result += vect1[i] \* vect2[i];

    }

    return result;

}

double \*ScalarVectMult(double \*vector, double scalar)

{

    double \*result = (double \*)malloc(N \* sizeof(double));

    for (int i = 0; i < N; i++)

    {

        result[i] = vector[i] \* scalar;

    }

    return result;

}

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

{

    int procSize;

    int procRank;

    MPI\_Init(&argc, &argv);

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

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

    // Matrix A initializing and sending parts to other processes

    // Matrix A generated on 0 process

    double \*A\_full = NULL;

    if (procRank == 0)

    {

        A\_full = (double \*)malloc(N \* N \* sizeof(double));

        for (int i = 0; i < N; i++)

        {

            for (int j = i; j < N; j++)

            {

                if (i == j)

                {

                    A\_full[i \* N + j] = (double)(rand() % 100) + 200;

                }

                else

                {

                    A\_full[i \* N + j] = A\_full[j \* N + i] = (double)(rand() % 100);

                }

            }

        }

    }

    double start = MPI\_Wtime();

    int count1 = N / procSize;

    int count2 = N % procSize;

    int \*rowsNum = (int \*)calloc(procSize, sizeof(int));

    // number of elements in each process part

    int \*sendcounts = (int \*)calloc(procSize, sizeof(int));

    for (int i = 0; i < procSize; i++)

    {

        rowsNum[i] = count1 + (i < count2);

        sendcounts[i] = rowsNum[i] \* N;

    }

    // i-ое значение в массиве displs определяет смещение относительно

    // начала sendbuf для данных, посылаемых процессу i

    int \*displs = (int \*)calloc(procSize, sizeof(int));

    displs[0] = 0;

    for (int i = 1; i < procSize; i++)

    {

        displs[i] = displs[i - 1] + sendcounts[i - 1];

    }

    double \*A\_part = (double \*)malloc(sizeof(double) \* sendcounts[procRank]);

    MPI\_Scatterv(A\_full, sendcounts, displs, MPI\_DOUBLE, A\_part, sendcounts[procRank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    // A\_part dimension

    int A\_p\_width = N;

    int A\_p\_height = rowsNum[procRank];

    // Vector b generating and sending

    // b is common for all processes

    double \*b = (double \*)malloc(N \* sizeof(double));

    if (procRank == 0)

    {

        for (int i = 0; i < N; i++)

        {

            b[i] = (double)(rand() % 100);

        }

    }

    // sending vector b to every process

    MPI\_Bcast(b, N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    // b dimension

    int b\_width = 1;

    int b\_height = N;

    double \*X = (double \*)calloc(N, sizeof(double));

    double \*r = (double \*)malloc(N \* sizeof(double));

    double \*prev\_r = (double \*)malloc(N \* sizeof(double));

    double \*z = (double \*)malloc(N \* sizeof(double));

    double square\_b\_len = 0; // square of vector b length

    square\_b\_len = VectorsMult(b, b);

    for (int i = 0; i < N; i++)

    {

        r[i] = b[i]; // r0 = b - Ax0, x0 - нулеой вектор

        z[i] = r[i]; // z0 = r0

    }

    double \*temp = (double \*)calloc(N, sizeof(double));

    double alpha, beta;

    int cntr = 0;

    int cnt = 0;

    double restrictor = epsilon \* epsilon \* square\_b\_len;

    do

    {

        // A\*z(n)

        // MatrixMultVector(A\_part, z, temp, A\_p\_height, A\_p\_width);

        parallelMatrixMultVector(A\_part, z, temp, A\_p\_width, A\_p\_height, procSize, procRank);

        // alpha(n+1) = (r(n), r(n)) / (Az(n), z(n))

        double oldsqrR = VectorsMult(r, r);

        alpha = oldsqrR / VectorsMult(temp, z);

        // X(n+1) = X(n) + alpha(n+1)\*z(n)

        X = SumVectors(X, ScalarVectMult(z, alpha));

        prev\_r = r;

        // r(n+1) = r(n) - alpha(n+1)Az(n)

        r = SubVectors(r, ScalarVectMult(temp, alpha));

        // beta(n+1) = (r(n+1), r(n+1)) / (r(n), r(n))

        double temp\_res = VectorsMult(r, r);

        beta = temp\_res / oldsqrR;

        // z(n+1) = r(n+1) + beta(n+1)z(n)

        z = SumVectors(r, ScalarVectMult(z, beta));

        if (temp\_res < restrictor)

        {

            cntr++;

        }

        cnt++;

    } while (cntr < 3);

    double finish = MPI\_Wtime();

    double total\_time = finish - start;

    if (procRank == 0)

    {

        printf("Iterations count: %d\n", cnt);

        printf("Total time: %f \n", total\_time);

    }

    free(A\_full);

    free(b);

    free(r);

    free(prev\_r);

    free(z);

    free(X);

    MPI\_Barrier(MPI\_COMM\_WORLD);

    MPI\_Finalize();

    return 0;

}

# Приложение 6. *Параллельная программа 2*

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include "mpi.h"

#define N 1400

#define epsilon 10e-5

// Matrix transpose isn’t required

void MatrixMultVector(double \*A, double \*vector, double \*result, int height, int width)

{

    for (int j = 0; j < width; j++)

    {

        for (int i = 0; i < height; i++)

        {

            result[i] += A[j \* height + i] \* vector[j];

        }

    }

}

void parallelMatrixMultVector(double \*A\_part, double \*vector, double \*result,

                              int A\_p\_width, int A\_p\_height, int procSize, int procRank)

{

    // A\_part is N \* rowsNum[procRank]

    // res is vector (one column matrix) of A\_part height

    double \*tmp = (double \*)calloc(A\_p\_height, sizeof(double));

    MatrixMultVector(A\_part, vector, tmp, A\_p\_height, A\_p\_width);

    double \*full\_result = NULL;

    if (procRank == 0)

    {

        full\_result = (double \*)calloc(A\_p\_height, sizeof(double));

    }

    MPI\_Reduce(tmp, full\_result, A\_p\_height, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

    // array of numbers of elemets send by each process

    int \*sendcounts = (int \*)calloc(procSize, sizeof(int));

    for (int i = 0; i < procSize; ++i)

    {

        sendcounts[i] = A\_p\_height / procSize;

        sendcounts[i] += (i < A\_p\_height % procSize);

    }

    int \*displs = (int \*)calloc(procSize, sizeof(int));

    displs[0] = 0;

    for (int i = 1; i < procSize; ++i)

    {

        displs[i] = displs[i - 1] + sendcounts[i - 1];

    }

    MPI\_Scatterv(full\_result, sendcounts, displs, MPI\_DOUBLE, result, sendcounts[procRank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    if (procRank == 0)

    {

        free(full\_result);

    }

}

double \*SubVectors(double \*vect1, double \*vect2, int size)

{

    double \*result = (double \*)malloc(size \* sizeof(double));

    for (int i = 0; i < size; i++)

    {

        result[i] = vect1[i] - vect2[i];

    }

    return result;

}

double \*SumVectors(double \*vect1, double \*vect2, int size)

{

    double \*result = (double \*)malloc(size \* sizeof(double));

    for (int i = 0; i < size; i++)

    {

        result[i] = vect1[i] + vect2[i];

    }

    return result;

}

double \*ScalarVectMult(double \*vector, double scalar, int size)

{

    double \*result = (double \*)malloc(size \* sizeof(double));

    for (int i = 0; i < size; i++)

    {

        result[i] = vector[i] \* scalar;

    }

    return result;

}

void VectorParallelMult(double \*vect1, double \*vect2, double \*res, int size)

{

    double tmp = 0;

    for (int i = 0; i < size; i++)

    {

        tmp += vect1[i] \* vect2[i];

    }

    MPI\_Allreduce(&tmp, res, 1, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

}

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

{

    MPI\_Init(&argc, &argv);

    int procSize;

    int procRank;

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

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

    // Matrix A initializing and sending parts to other processes

    // Matrix A generated on 0 process

    double \*A\_full = NULL;

    if (procRank == 0)

    {

        A\_full = (double \*)malloc(N \* N \* sizeof(double));

        for (int i = 0; i < N; i++)

        {

            for (int j = i; j < N; j++)

            {

                if (i == j)

                {

                    A\_full[i \* N + j] = (double)(rand() % 100) + 200;

                }

                else

                {

                    A\_full[i \* N + j] = A\_full[j \* N + i] = (double)(rand() % 100);

                }

            }

        }

    }

    double start = MPI\_Wtime();

    // number of elements in each process part

    int \*sendcounts = (int \*)calloc(procSize, sizeof(int));

    for (int i = 0; i < procSize; i++)

    {

        sendcounts[i] = N / procSize + (i < N % procSize);

        sendcounts[i] \*= N;

    }

    // i-ое значение в массиве displs определяет смещение относительно

    // начала sendbuf для данных, посылаемых процессу i

    int \*displs = (int \*)calloc(procSize, sizeof(int));

    displs[0] = 0;

    for (int i = 1; i < procSize; i++)

    {

        displs[i] = displs[i - 1] + sendcounts[i - 1];

    }

    double \*temp\_part = (double \*)malloc(sizeof(double) \* sendcounts[procRank]);

    MPI\_Scatterv(A\_full, sendcounts, displs, MPI\_DOUBLE, temp\_part, sendcounts[procRank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    // A\_part dimension

    // int A\_p\_width = rowsNum[procRank];

    int A\_p\_width = sendcounts[procRank] / N;

    int A\_p\_height = N;

    double \*A\_part = (double \*)malloc(sizeof(double) \* sendcounts[procRank]);

    A\_part = temp\_part;

    // generating vector b on zero process

    double \*b\_full = NULL;

    if (procRank == 0)

    {

        b\_full = (double \*)malloc(sizeof(double) \* N);

        for (int i = 0; i < N; i++)

        {

            b\_full[i] = (double)(rand() % 100);

        }

    }

    for (int i = 0; i < procSize; i++)

    {

        sendcounts[i] /= N;

    }

    displs[0] = 0;

    for (int i = 1; i < procSize; i++)

    {

        displs[i] = displs[i - 1] + sendcounts[i - 1];

    }

    double \*b\_part = (double \*)malloc(sizeof(double) \* sendcounts[procRank]);

    // vector b scattering

    MPI\_Scatterv(b\_full, sendcounts, displs, MPI\_DOUBLE, b\_part, sendcounts[procRank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    // b dimension

    int b\_width = 1;

    int b\_part\_height = sendcounts[procRank];

    // int b\_height = N;

    double \*X = (double \*)calloc(b\_part\_height, sizeof(double));

    double \*r = (double \*)malloc(b\_part\_height \* sizeof(double));

    double \*prev\_r = (double \*)malloc(b\_part\_height \* sizeof(double));

    double \*z = (double \*)malloc(b\_part\_height \* sizeof(double));

    // square of vector b length

    double square\_b\_len = 0;

    VectorParallelMult(b\_part, b\_part, &square\_b\_len, b\_part\_height);

    for (int i = 0; i < b\_part\_height; i++)

    {

        r[i] = b\_part[i]; // r0 = b - Ax0, x0 - нулеой вектор

        z[i] = r[i];      // z0 = r0

    }

    int counter = 0;

    double tmp\_scalar = 0;

    double r\_len = 0;

    double \*temp = (double \*)calloc(b\_part\_height, sizeof(double));

    double alpha, beta;

    int cntr = 0;

    int cnt = 0;

    double restrictor = epsilon \* epsilon \* square\_b\_len;

    do

    {

        // temp = A\*z(n)

        // MatrixMultVector(A\_part, z, temp, A\_p\_height, A\_p\_width);

        parallelMatrixMultVector(A\_part, z, temp, A\_p\_width, A\_p\_height, procSize, procRank);

        // alpha(n+1) = (r(n), r(n)) / (Az(n), z(n))

        VectorParallelMult(r, r, &r\_len, b\_part\_height);

        VectorParallelMult(temp, z, &tmp\_scalar, b\_part\_height);

        // double oldsqrR = VectorParallelMult(r, r, b\_part\_height);

        alpha = r\_len / tmp\_scalar;

        // X(n+1) = X(n) + alpha(n+1)\*z(n)

        X = SumVectors(X, ScalarVectMult(z, alpha, b\_part\_height), b\_part\_height);

        prev\_r = r;

        // r(n+1) = r(n) - alpha(n+1)Az(n)

        r = SubVectors(r, ScalarVectMult(temp, alpha, b\_part\_height), b\_part\_height);

        // beta(n+1) = (r(n+1), r(n+1)) / (r(n), r(n))

        double temp\_res = 0;

        VectorParallelMult(r, r, &temp\_res, b\_part\_height);

        beta = temp\_res / r\_len;

        // z(n+1) = r(n+1) + beta(n+1)z(n)

        z = SumVectors(r, ScalarVectMult(z, beta, b\_part\_height), b\_part\_height);

        if (temp\_res < restrictor)

        {

            cntr++;

        }

        cnt++;

    } while (cntr < 3);

    double finish = MPI\_Wtime();

    double total\_time = finish - start;

    if (procRank == 0)

    {

        printf("Iterations count: %d\n", cnt);

        printf("Total time: %f \n", total\_time);

    }

    double \*result = (double \*)calloc(N, sizeof(double));

    MPI\_Gatherv(X, sendcounts[procRank], MPI\_DOUBLE, result, sendcounts, displs, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

    free(A\_full);

    free(b\_full);

    free(r);

    free(prev\_r);

    free(z);

    free(X);

    MPI\_Barrier(MPI\_COMM\_WORLD);

    MPI\_Finalize();

    return 0;

}