**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**

«ПАРАЛЛЕЛЬНАЯ РЕАЛИЗАЦИЯ РЕШЕНИЯ СИСТЕМЫ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИХ УРАВНЕНИЙ С ПОМОЩЬЮ MPI»

студентки 2 курса, группы 22204

**Клочихиной Софьи Павловны**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А. Ю. Власенко

Новосибирск 2024

**СОДЕРЖАНИЕ**

ЦЕЛЬ 3

ЗАДАНИЕ 4

ОПИСАНИЕ РАБОТЫ 5

ЗАКЛЮЧЕНИЕ 8

ПРИЛОЖЕНИЕ.

Полный листинг последовательной программы на C++ 9

Полный листинг параллельной программы на C++ 12

**ЦЕЛЬ**

1. Сравнить время выполнения решения системы линейных алгебраических уравнений вида методом простых итераций последовательной и параллельной программ.

**ЗАДАНИЕ**

1. Написать 2 программы (последовательную и параллельную с использованием MPI) на языке C/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. Параллельную программу реализовать с тем условием, что матрица *A* и вектор *b* инициализируются на каком-либо одном процессе, а затем матрица *A* «разрезается» по строкам на близкие по размеру, возможно не одинаковые, части, а вектор *b* раздается каждому процессу.  
Уделить внимание тому, чтобы при запуске программы на различном  
числе MPI-процессов решалась одна и та же задача (исходные данные  
заполнялись одинаковым образом).

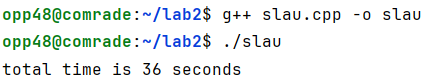
3. Замерить время работы последовательного варианта программы, а также время работы параллельного при использовании различного числа процессорных ядер. Минимально на 2, 4, 8, 16, 24 (на каждом из наших узлов кластера по 12 ядер), но чем на большем количестве различного числа процессов будут выполнены замеры, тем лучше. Также чем больше замеров будет выполнено на одном и том же числе 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 или 24-х ядер.

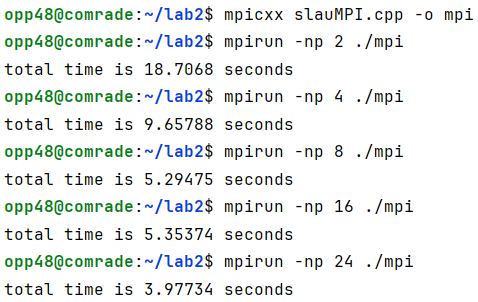
5. На основании полученных результатов сделать вывод.

**ОПИСАНИЕ РАБОТЫ**

1. На языке программирования С++ были написаны 2 программы: последовательная и параллельная, в которой использовалась библиотека MPI, позволяющая сделать программу параллельной. Время выполнения второй было замерено с помощью функции MPI\_Wtime();
2. Компиляция, запуск и выходные данные последовательной программы выглядели следующим образом:



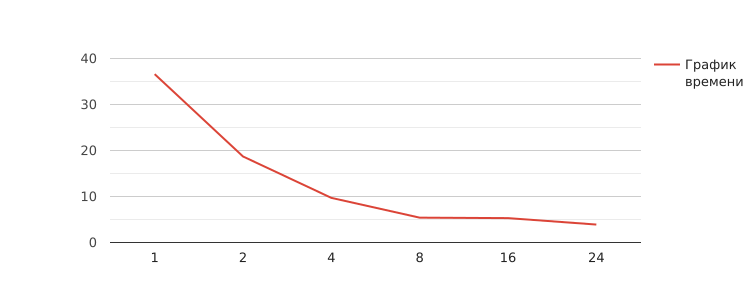
1. Компиляция, запуск и минимальные выходные данные параллельной программы:

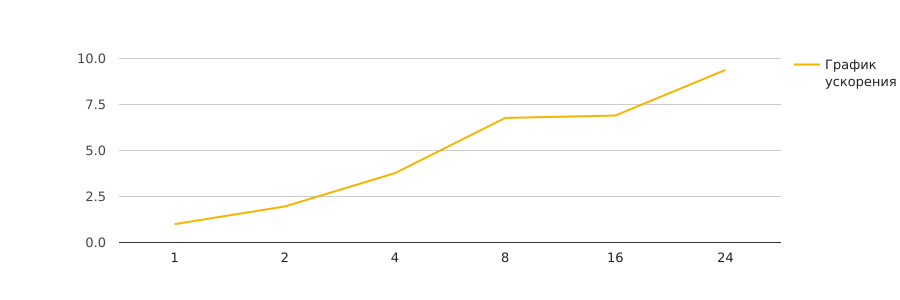


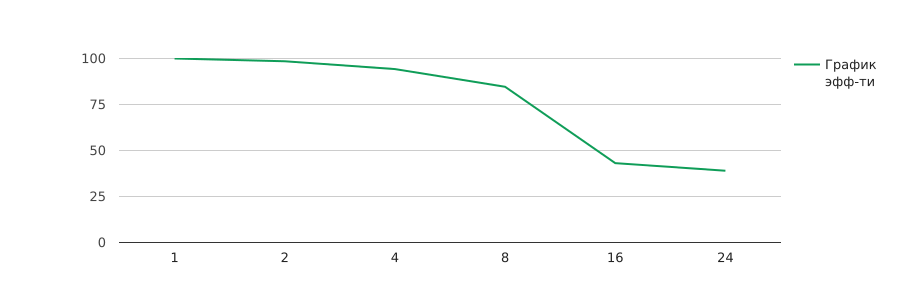
1. Было выполнено построение графиков времени, ускорения и эффективности, где:

**Ускорение**: Sp = T1 / Tp, где T1 - время работы ПОСЛЕДОВАТЕЛЬНОЙ программы. Tp - время работы параллельной программы на p процессах/потоках.

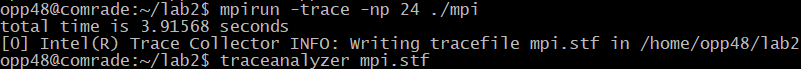
**Эффективность**: Ep = Sp / p \* 100%.

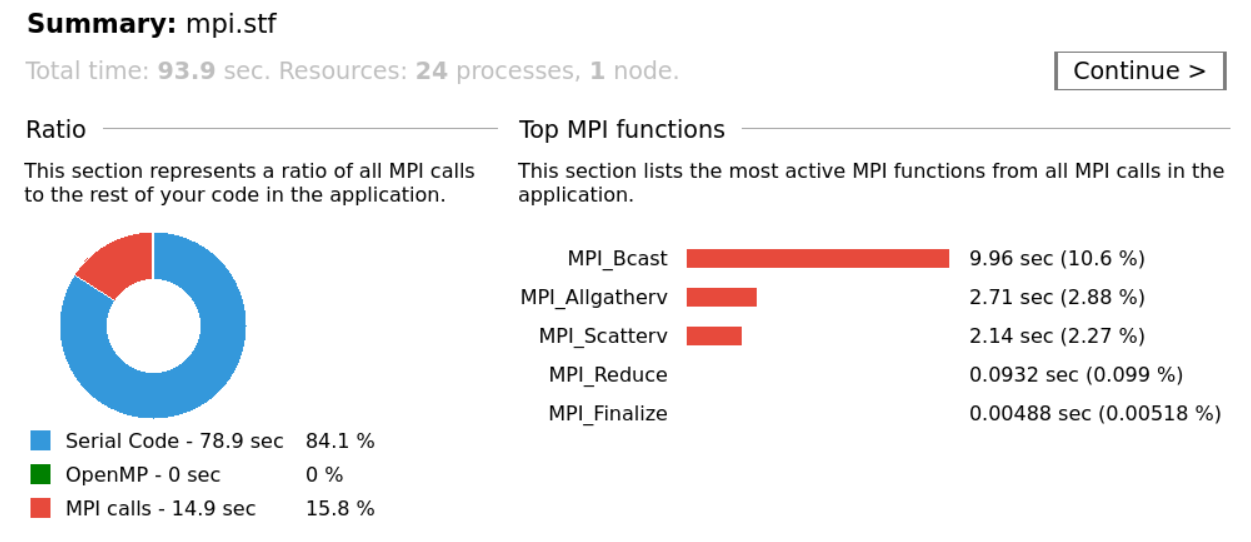


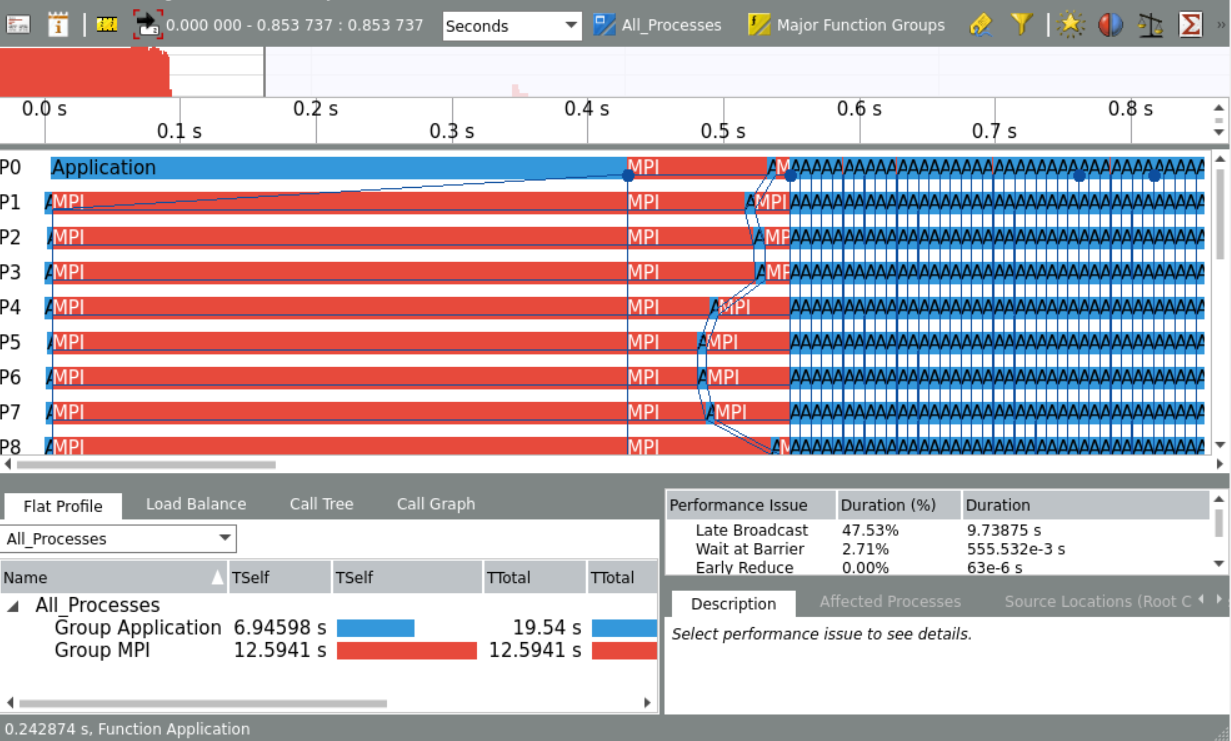




1. В конце было выполнено профилирование:







**ЗАКЛЮЧЕНИЕ**

Распараллеленная программа работает в несколько раз быстрее последовательной за счёт того, что процессы обрабатывают свои данные одновременно и распределёнными частями.

1. **Приложение 1.** Полный листинг последовательной программы на C++.

#include <iostream>

#include <cmath>

const double EPSILON = 0.0001;

const int N = 4000;

const float TAU = 0.00001;

const int MAX\_ITERATION\_NUM = 10000;

void createMatrix(double\* matrix) {

for (int i = 0; i < N; i++) {

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

matrix[i \* N + j] = (double)rand() / RAND\_MAX \* 2.0 - 1.0;

if (i == j)

matrix[i \* N + j] = matrix[i \* N + j] + N;

}

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

matrix[i \* N + j] = matrix[j \* N + i];

}

}

void createVector(double\* vector) {

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

vector[i] = (double)rand() / RAND\_MAX \* 10.0 - 5.0;

}

double countAccuracy(const double\* vector) {

double accuracy = 0;

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

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

return sqrt(accuracy);

}

void simpleIteration(const double\* A, double\* b, double\* x) {

auto\* x\_new = new double[N];

int iter\_count = 0;

double accuracy = 1;

while (accuracy > EPSILON && iter\_count < MAX\_ITERATION\_NUM) {

for (int i = 0; i < N; i++) {

x\_new[i] = 0;

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

x\_new[i] += A[i \* N + j] \* x[j];

}

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

x\_new[i] = x\_new[i] - b[i];

accuracy = countAccuracy(x\_new) / countAccuracy(b);

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

x[i] = x[i] - TAU \* x\_new[i];

iter\_count++;

}

delete[] x\_new;

}

int main() {

srand(time(nullptr));

auto\* A = new double[N \* N];

auto\* x = new double[N];

auto\* b = new double[N];

createMatrix(A);

createVector(x);

createVector(b);

time\_t begin = time(nullptr);

simpleIteration(A, b, x);

time\_t end = time(nullptr);

std::cout << "total time is " << end - begin << " seconds" << std::endl;

delete[] A;

delete[] x;

delete[] b;

return 0;

}

**Приложение 2.** Полный листинг параллельной программы на C++.

#include <iostream>

#include <cmath>

#include <mpi.h>

const double EPSILON = 0.0001;

const int N = 4000;

const float TAU = 0.00001;

const int MAX\_ITERATION\_NUM = 10000;

void createMatrix(double\* matrix) {

for (int i = 0; i < N; i++) {

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

matrix[i \* N + j] = (double)rand() / RAND\_MAX \* 2.0 - 1.0;

if (i == j)

matrix[i \* N + j] = matrix[i \* N + j] + N;

}

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

matrix[i \* N + j] = matrix[j \* N + i];

}

}

void createVector(double\* vector) {

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

vector[i] = (double)rand() / RAND\_MAX \* 10.0 - 5.0;

}

void setMatrixParts(int\* linesPerProc, int\* sendCounts, int\* displs, int\* offsets, int size){

int lines = N / size, rest = N % size, offset = 0;

for (int i = 0; i < size; ++i) {

rest ? (linesPerProc[i] = lines + 1, rest--) : linesPerProc[i] = lines;

offsets[i] = offset;

displs[i] = offset \* N;

sendCounts[i] = linesPerProc[i] \* N;

offset += linesPerProc[i];

}

}

double countPartAccuracy(const double\* vector, int end) {

double accuracy = 0;

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

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

return accuracy;

}

void simpleIteration(const double\* A\_part, double\* b, double\* x, const int\* linesPerProc, const int\* displs, const int\* offsets, int rank) {

auto\* x\_new = new double[linesPerProc[rank]];

int iter\_count = 0;

double accuracy = 1;

double part\_accuracy;

double B\_accuracy;

if (!rank) B\_accuracy = sqrt(countPartAccuracy(b, N));

while (accuracy > EPSILON && iter\_count < MAX\_ITERATION\_NUM) {

for (int i = 0; i < linesPerProc[rank]; i++) {

x\_new[i] = 0;

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

x\_new[i] += A\_part[i \* N + j] \* x[j];

}

for (int i = 0; i < linesPerProc[rank]; i++)

x\_new[i] = x\_new[i] - b[offsets[rank] + i];

part\_accuracy = countPartAccuracy(x\_new, linesPerProc[rank]);

for (int i = 0; i < linesPerProc[rank]; i++)

x\_new[i] = x[offsets[rank] + i] - TAU \* x\_new[i];

MPI\_Allgatherv(x\_new, linesPerProc[rank], MPI\_DOUBLE,

x, linesPerProc, offsets, MPI\_DOUBLE, MPI\_COMM\_WORLD);

if (!rank) accuracy = 0;

MPI\_Reduce(&part\_accuracy, &accuracy, 1,MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

if (!rank) {

accuracy = sqrt(accuracy) / B\_accuracy;

iter\_count++;

}

MPI\_Bcast(&iter\_count, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

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

}

delete[] x\_new;

}

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

srand(time(nullptr));

int rank, size;

double begin, end;

MPI\_Init(&argc, &argv);

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

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

auto\* linesPerProc = new int[size];

auto\* offsets = new int[size];

auto\* sendCounts = new int[N];

auto\* displs = new int[N];

double\* A;

setMatrixParts(linesPerProc, sendCounts, displs, offsets, size);

auto\* x = new double[N];

auto\* b = new double[N];

if (!rank) {

begin = MPI\_Wtime();

A = new double[N \* N];

createMatrix(A);

createVector(x);

createVector(b);

}

auto\* A\_part = new double[linesPerProc[rank] \* N];

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

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

MPI\_Scatterv(A, sendCounts, displs, MPI\_DOUBLE,

A\_part, sendCounts[rank], MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

simpleIteration(A\_part, b, x, linesPerProc, displs, offsets, rank);

delete[] A\_part;

delete[] x;

delete[] b;

delete[] linesPerProc;

delete[] sendCounts;

delete[] displs;

delete[] offsets;

if (!rank) {

delete[] A;

end = MPI\_Wtime();

std::cout << "total time is " << end - begin << " seconds" << std::endl;

}

MPI\_Finalize();

return 0;

}