###### 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»

студентки 2 курса, 19202 группы

**Хаецкой Дарьи Владимировны**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.Ю. Власенко

Новосибирск 2021

**СОДЕРЖАНИЕ**

[ЗАДАНИЕ 3](#_heading=h.gjdgxs)

[ОПИСАНИЕ РАБОТЫ 4](#_heading=h.30j0zll)

[Приложение 1 6](#_heading=h.1fob9te)

Приложение 2 12

Приложение 3 16

# ЗАДАНИЕ

1. Написать программу на языке C или C++, которая реализует 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. Программу распараллелить с помощью MPI с разрезанием матрицы A по строкам на близкие по размеру, возможно не одинаковые, части. Соседние строки матрицы должны располагаться в одном или в соседних MPI-процессах. Реализовать два варианта программы:

* Вариант 1: векторы x и b дублируются в каждом MPI-процессе,
* Вариант 2: векторы x и b разрезаются между MPI-процессами аналогично матрице A. Уделить внимание тому, чтобы при запуске программы на различном числе MPI-процессов решалась одна и та же задача (исходные данные заполнялись одинаковым образом).

1. Замерить время работы двух вариантов программы при использовании различного числа процессорных ядер: 1,2, 4, 8, 16. 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.
2. Выполнить профилирование двух вариантов программы с помощью MPE при использовании 16-и ядер.
3. На основании полученных результатов сделать вывод о целесообразности использования одного или второго варианта программы

# 

# ОПИСАНИЕ РАБОТЫ

# В ходе работы был реализован метод простой итерации, написана последовательная программа, затем распараллеленная с разделением векторов х и b, и без него. В ходе анализа времени работы программы, а так же зависимости ускорения и эффективности программы, было выяснено, что увеличение числа ядер после определённого значения уже не даёт значительного прироста скорости и не является оправданным

# Профилирование:

Для варианта 1:

# 

# 

# Для варианта 2:

# 

# Приложение 1. *Листинг программы вариант 1*

#include <iostream>

#include <cmath>

#include <mpi.h>

const long int N = 6400;

const double epsilon = pow(10, -5);

const double parameter = 0.0001;

double EuclideanNorm(const double\* u){

double norm = 0;

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

norm += u[i]\*u[i];

}

return sqrt(norm);

}

void sub(double\* a, double\* b, double\* c, int n){

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

c[i] = a[i] - b[i];

}

}

void mul(double\* A, double\* b, double\* result, int n) {

for(unsigned int i = 0; i < n; i++){

result[i] = 0;

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

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

}

}

}

void scalMul(double\* A, double tau, int n){

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

A[i] = A[i] \* tau;

}

}

double drand(double low, double high) {

double f = (double)rand() / RAND\_MAX;

return low + f \* (high - low);

}

double rand\_double(){

return (double)rand()/RAND\_MAX\*4.0 - 2.0;

}

void generate\_matrix(double\* matrix) {

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

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

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

}

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

matrix[i\*N + j] = rand\_double();

if(i == j){

matrix[i\*N + j] = fabs(matrix[i\*N + j]) + 400.0;

}

}

}

}

void printVector(const double\* B, const char\* name, int procRank, int procNum, int n){

for (int numProc = 0; numProc < procNum; ++numProc) {

if (procRank == numProc) {

printf("%s in rank %d:\n", name, procRank);

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

printf("%lf\n", B[i]);

}

printf("\n");

}

}

}

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

srand(time(0));

int process\_Rank, size\_Of\_Cluster;

MPI\_Init(&argc, &argv);

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

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &process\_Rank);

// process shared buffers

double\* ABuf = (double\*)malloc(N\*N / size\_Of\_Cluster \* sizeof(double)); // free

double\* AxBuf = (double\*)malloc(N / size\_Of\_Cluster \* sizeof(double)); // free

double\* prevX = nullptr;

double\* Ax = nullptr;

double\* nextX = nullptr;

double\* A = nullptr;

double\* b = nullptr;

double tau = parameter;

double startTime = 0, endTime = 0, currentTime = 0;

bool timeOut = false;

int timeLimit = 120;

double normAxb = 0; // ||A\*xn - b||

double normb = 0;

double saveRes = 1;

double res = 1;

double lastres = 1;

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

prevX = (double\*)malloc(N \* sizeof(double));// free

Ax = (double\*)malloc(N \* sizeof(double));// free

nextX = (double\*)malloc(N \* sizeof(double));// free

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

prevX[i] = drand(1,5);

nextX[i] = drand(1,5);

b[i] = drand(1,N);

}

if (process\_Rank == 0){

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

generate\_matrix(A);

mul(A, prevX, Ax, N); // A\*xn

normb = EuclideanNorm(b);

}

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

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

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

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

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

MPI\_Scatter(A, N \* N / size\_Of\_Cluster, MPI\_DOUBLE, ABuf,

N \* N / size\_Of\_Cluster, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

int countIt = 1;

startTime = MPI\_Wtime();

while (res > epsilon){

if(process\_Rank == 0) {

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

prevX[i] = nextX[i];

}

}

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

MPI\_Scatter(Ax, N / size\_Of\_Cluster, MPI\_DOUBLE, AxBuf,

N / size\_Of\_Cluster, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

mul(ABuf, prevX, AxBuf, N / size\_Of\_Cluster); // A\*xn

sub(AxBuf, b, AxBuf, N / size\_Of\_Cluster); // A\*xn - b

scalMul(AxBuf, tau, N / size\_Of\_Cluster); // TAU\*(A\*xn - b)

MPI\_Allgather(AxBuf, N / size\_Of\_Cluster, MPI\_DOUBLE, Ax,

N / size\_Of\_Cluster, MPI\_DOUBLE, MPI\_COMM\_WORLD);

if(process\_Rank == 0){

normAxb = EuclideanNorm(Ax); // ||A\*xn - b||

sub(prevX, Ax, nextX, N); // xn - TAU \* (A\*xn - b)

res = normAxb / normb;

countIt++;

if ((countIt > 100000 && lastres > res) || res == INFINITY) {

if (tau < 0) {

printf("Does not converge\n");

saveRes = res;

res = 0;

}

else {

tau = (-1)\*parameter;

countIt = 0;

}

}

lastres = res;

}

currentTime = MPI\_Wtime();

if ((currentTime - startTime) > timeLimit){

timeOut = true;

}

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

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

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

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

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

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

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

MPI\_Barrier(MPI\_COMM\_WORLD);

}

endTime = MPI\_Wtime();

if (process\_Rank == 0){

if (res == 0){

res = saveRes;

}

if (!timeOut){

printf("%ld\*%ld matrix error coefficient is %lf, iterations: %d\n",N, N, res, countIt);

printf("That took %lf seconds\n",endTime-startTime);

} else {

printf("it took more than %d seconds so I killed the process,"

"error coefficient was %lf\n", timeLimit, res);

}

free(A);

}

free(ABuf);

free(Ax);

free(nextX);

free(prevX);

free(b);

free(AxBuf);

MPI\_Finalize();

return 0;

}

# Приложение 2. *Листинг программы вариант 2*

#include <iostream>

#include <cmath>

#include <mpi.h>

const long int N = 6400;

const double epsilon = pow(10, -5);

const double parameter = 0.0001;

double EuclideanNorm(const double\* u){

double norm = 0;

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

norm += u[i]\*u[i];

}

return sqrt(norm);

}

void sub(double\* a, double\* b, double\* c, int n){

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

c[i] = a[i] - b[i];

}

}

void InitialMul(double\* A, double\* b, double\* result, int n) {

for(unsigned int i = 0; i < n; i++){

result[i] = 0;

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

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

}

}

}

void mul(double\* A, double\* vec, double\* result, int rowcount) {

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

result[i] = 0;

for (int j = 0; j < rowcount; ++j){

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

}

}

}

void scalMul(double\* A, double tau, int n){

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

A[i] = A[i] \* tau;

}

}

double drand(double low, double high) {

double f = (double)rand() / RAND\_MAX;

return low + f \* (high - low);

}

double rand\_double(){

return (double)rand()/RAND\_MAX\*4.0 - 2.0;

}

void generate\_matrix(double\* matrix) {

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

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

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

}

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

matrix[i\*N + j] = rand\_double();

if(i == j){

matrix[i\*N + j] = fabs(matrix[i\*N + j]) + 400.0;

}

}

}

}

void printVector(const double\* B, const char\* name, int procRank, int procNum, int n){

for (int numProc = 0; numProc < procNum; ++numProc) {

if (procRank == numProc) {

printf("%s in rank %d:\n", name, procRank);

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

printf("%lf\n", B[i]);

}

printf("\n");

}

}

}

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

srand(time(0));

int process\_Rank, size\_Of\_Cluster;

MPI\_Init(&argc, &argv);

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

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &process\_Rank);

// process shared buffers

double\* ABuf = (double\*)malloc(N\*N / size\_Of\_Cluster \* sizeof(double));

double\* xBuf = (double\*)malloc(N / size\_Of\_Cluster \* sizeof(double));

double\* nextXBuf = (double\*)malloc(N / size\_Of\_Cluster \* sizeof(double));

double\* AxBuf = (double\*)malloc(N / size\_Of\_Cluster \* sizeof(double));

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

double\* bBuf = (double\*)malloc(N / size\_Of\_Cluster \* sizeof(double));

double\* prevX = nullptr;

double\* Ax = nullptr;

double\* nextX = nullptr;

double\* A = nullptr;

double\* b = nullptr;

double tau = parameter;

double startTime = 0, endTime = 0, currentTime = 0;

bool timeOut = false;

int timeLimit = 70;

double normAxb = 0; // ||A\*xn - b||

double normb = 0;

double saveRes = 1;

double res = 1;

double lastres = 1;

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

if (process\_Rank == 0){

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

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

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

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

prevX[i] = drand(1,5);

nextX[i] = drand(1,5);

b[i] = drand(1,N);

}

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

generate\_matrix(A);

InitialMul(A, prevX, Ax, N); // A\*xn

normb = EuclideanNorm(b);

}

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

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

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

MPI\_Scatter(A, N \* N / size\_Of\_Cluster, MPI\_DOUBLE, ABuf,

N \* N / size\_Of\_Cluster, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Scatter(b, N / size\_Of\_Cluster, MPI\_DOUBLE, bBuf,

N / size\_Of\_Cluster, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

int countIt = 1;

startTime = MPI\_Wtime();

while (res > epsilon && !timeOut){

MPI\_Scatter(prevX, N / size\_Of\_Cluster, MPI\_DOUBLE, xBuf,

N / size\_Of\_Cluster, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

mul(ABuf, xBuf, AxMulRes, N / size\_Of\_Cluster); // A\*xn

MPI\_Allreduce(AxMulRes, Ax, N, MPI\_DOUBLE, MPI\_SUM, MPI\_COMM\_WORLD);

MPI\_Scatter(Ax, N / size\_Of\_Cluster, MPI\_DOUBLE, AxBuf,

N / size\_Of\_Cluster, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

sub(AxBuf, bBuf, AxBuf, N / size\_Of\_Cluster); // A\*xn - b

scalMul(AxBuf, tau, N / size\_Of\_Cluster); // TAU\*(A\*xn - b)

sub(xBuf, AxBuf, nextXBuf, N / size\_Of\_Cluster); // xn - TAU \* (A\*xn - b)

MPI\_Gather(nextXBuf, N/size\_Of\_Cluster, MPI\_DOUBLE, prevX,

N/size\_Of\_Cluster, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

MPI\_Gather(AxBuf, N/size\_Of\_Cluster, MPI\_DOUBLE, Ax,

N/size\_Of\_Cluster, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

if(process\_Rank == 0){

normAxb = EuclideanNorm(Ax); // ||A\*xn - b||

res = normAxb / normb;

countIt++;

if ((countIt > 100000 && lastres > res) || res == INFINITY) {

// if (countIt > 4){

if (tau < 0) {

printf("Does not converge\n");

saveRes = res;

res = 0;

}

else {

tau = (-1)\*parameter;

countIt = 0;

}

}

lastres = res;

}

currentTime = MPI\_Wtime();

if ((currentTime - startTime) > timeLimit){

timeOut = true;

}

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

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

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

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

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

}

endTime = MPI\_Wtime();

if (process\_Rank == 0){

if (res == 0){

res = saveRes;

}

if (!timeOut){

printf("%ld\*%ld matrix error coefficient is %lf, iterations: %d\n",N, N, res, countIt);

printf("That took %lf seconds\n",endTime-startTime);

} else {

printf("it took more than %d seconds so I killed the proccess,"

"error coefficient was %lf\n", timeLimit, res);

}

free(A);

free(b);

free(nextX);

free(prevX);

free(Ax);

}

free(nextXBuf);

free(xBuf);

free(ABuf);

free(AxBuf);

free(AxMulRes);

free(bBuf);

MPI\_Finalize();

return 0;

}

# Приложение 3. *Листинг последовательной программы*

#include <iostream>

#include <cmath>

const long int N = 6400;

const double epsilon = pow(10, -5);

const float parameter = 0.0001;

double EuclideanNorm(const double\* u){

double norm = 0;

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

norm += u[i]\*u[i];

}

return sqrt(norm);

}

void sub(double\* a, double\* b, double\* c){

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

c[i] = a[i] - b[i];

}

}

void mul(double\* A, double\* b, double\* result, int n) {

for(unsigned int i = 0; i < n; i++) {

result[i] = 0;

for(unsigned int j = 0; j < n; j++) {

result[i] += A[i \* n + j] \* b[j];

}

}

}

void scalMul(double\* A, double tau){

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

A[i] = A[i] \* tau;

}

}

void printMatrix(double\* A){

printf("\n");

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

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

printf("%lf ", A[i \* N + j]);

}

printf("\n");

}

printf("\n");

}

void printVec(double\* A, int n){

printf("\n");

printf("\n");

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

printf("%lf ", A[i]);

}

printf("\n");

printf("\n");

}

double drand(double low, double high) {

double f = (double)rand() / RAND\_MAX;

return low + f \* (high - low);

}

double rand\_double(){

return (double)rand()/RAND\_MAX\*4.0 - 2.0;

}

void generate\_matrix(double\* matrix) {

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

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

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

}

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

matrix[i\*N + j] = rand\_double();

if(i == j){

matrix[i\*N + j] = fabs(matrix[i\*N + j]) + 400.0;

}

}

}

}

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

srand(time(0));

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

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

double tau = parameter;

double\* b = nullptr;

double\* nextX = nullptr;

double\* A = nullptr;

double normAxb = 0; // ||A\*xn - b||

double normb = 0;

double saveRes = 0;

double res = 0;

double lastres = 0;

clock\_t timeLimit = 600;

bool timeOut = false;

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

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

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

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

prevX[i] = drand(1, 5);

b[i] = drand(1, 5);

}

generate\_matrix(A);

mul(A, prevX, Ax, N); // A\*xn

sub(Ax, b, Ax); // A\*xn - b

normAxb = EuclideanNorm(Ax); // ||A\*xn - b||

normb = EuclideanNorm(b);

scalMul(Ax, tau); // TAU\*(A\*xn - b)

sub(prevX, Ax, nextX); // xn - TAU \* (A\*xn - b)

saveRes = normAxb / normb;

res = normAxb / normb;

lastres = res;

int countIt = 1;

clock\_t start, currentTime, end;

double cpu\_time\_used;

start = clock();

while (res > epsilon) {

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

prevX[i] = nextX[i];

}

mul(A, prevX, Ax, N); //A\*xn

sub(Ax, b, Ax); //A\*xn - b

normAxb = EuclideanNorm(Ax); // ||A\*xn - b||

scalMul(Ax, tau); // TAU\*(A\*xn - b)

sub(prevX, Ax, nextX); // xn - TAU \* (A\*xn - b)

res = normAxb / normb;

countIt++;

if ((countIt > 100000 && lastres > res) || res == INFINITY) {

if (tau < 0) {

printf("Does not converge\n");

free(Ax);

free(nextX);

free(prevX);

free(b);

free(A);

return 0;

}

else {

tau = -0.01;

countIt = 0;

res = saveRes;

}

}

lastres = res;

currentTime = clock();

if ( ((double) (currentTime - start)) / CLOCKS\_PER\_SEC > timeLimit){

res = 0;

timeOut = true;

}

}

end = clock();

if (!timeOut){

printf("%ld\*%ld matrix error coefficient is %lf, iterations: %d\n",N, N, res, countIt);

printf("That took %lf seconds\n", ((double) (end - start)) / CLOCKS\_PER\_SEC);

} else {

printf("it took more than %ld seconds so I killed the process\n", timeLimit);

}

free(Ax);

free(nextX);

free(prevX);

free(b);

free(A);

return 0;

}