Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»

Институт информационных технологий, математики и механики

Кафедра математического обеспечения и суперкомпьютерных технологий

**ОТЧЕТ**

по предмету «Анализ производительности и оптимизация программного обеспечения»

**Выполнил**:

Студент группы 382003-3м

Вдовин Евгений Александрович

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись

**Проверил**:

Доцент кафедры МОСТ, к.т.н.

Мееров Иосиф Борисович

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись

Нижний Новгород

2020

**Содержание**

[**Постановка задачи** 3](#_Toc59980779)

[**Описание базовой версии кода и анализ производительности** 4](#_Toc59980780)

[**Процесс оптимизации** 7](#_Toc59980781)

[**Заключение** 8](#_Toc59980782)

[**Приложение. Код программы** 9](#_Toc59980783)

[**Problem.h** 9](#_Toc59980784)

[**Problem.cpp** 9](#_Toc59980785)

[**SupportStructures.h** 10](#_Toc59980786)

[**App.cpp** 11](#_Toc59980787)

# **Постановка задачи**

Задачи минимизации (или максимизации) функций при различных дополнительных условиях являются типичными математическими моделями процессов выбора решений при автоматизированном проектировании технических устройств и систем, в управлении подвижными частями роботов, при восстановлении зависимостей на основе анализа экспериментальных данных и т.д.

Алгоритм глобального поиска предназначен для отыскания точек и значений абсолютного минимума действительной функции на отрезке [a,b] вещественной оси :

Вычислительная схема алгоритма:

1. Первые два испытания осуществляются в точках .
2. Перенумеровать нижним индексом точки в порядке возрастания значения координаты.
3. Найти , где ,
4. Положить , где .
5. Для каждого интервала , вычислить
6. Найти , при котором .
7. Положить .

C помощью данного алгоритма добавляем точки в список найденных точек. Далее выполняем добавление точек, пока не достигнем требуемой точности (, или пока не будет выполнено заданное число испытаний.

Необходимо провести анализ производительности кода, реализующего этот алгоритм и провести процедуру его оптимизации.

# **Описание базовой версии кода и анализ производительности**

Базовая версия кода содержит реализацию алгоритма глобального поиска – функция AGSLinearVersion.

Для теста была выбрана функция Schwefel, на которой и искался оптимум.

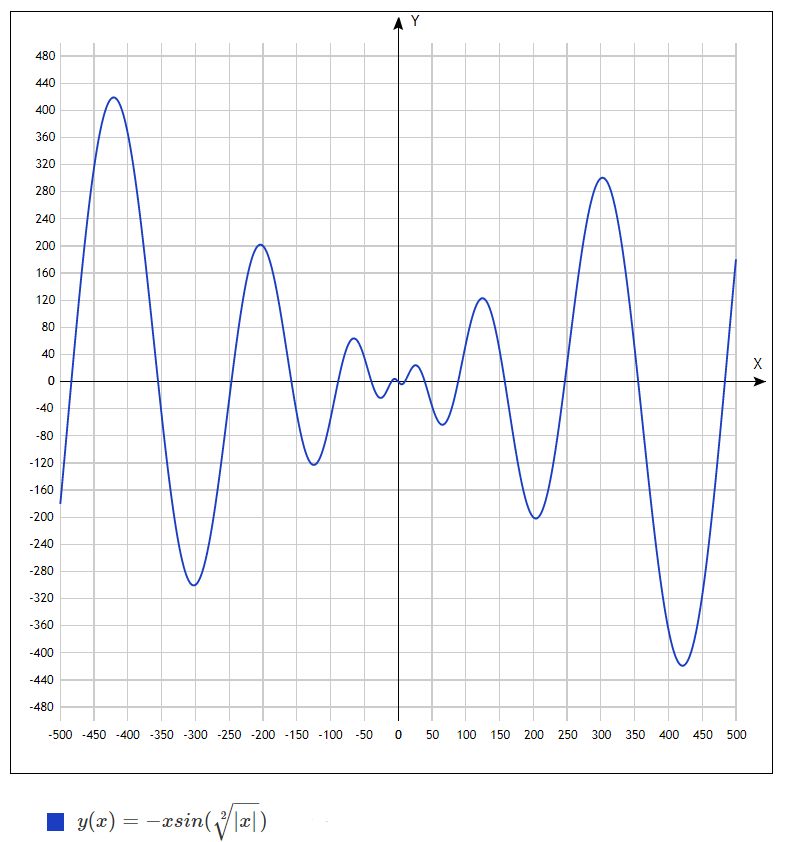


Рис. 1. График функции Schwefel в одномерном случае

Для этой функции известен оптимум и границы его поиска:

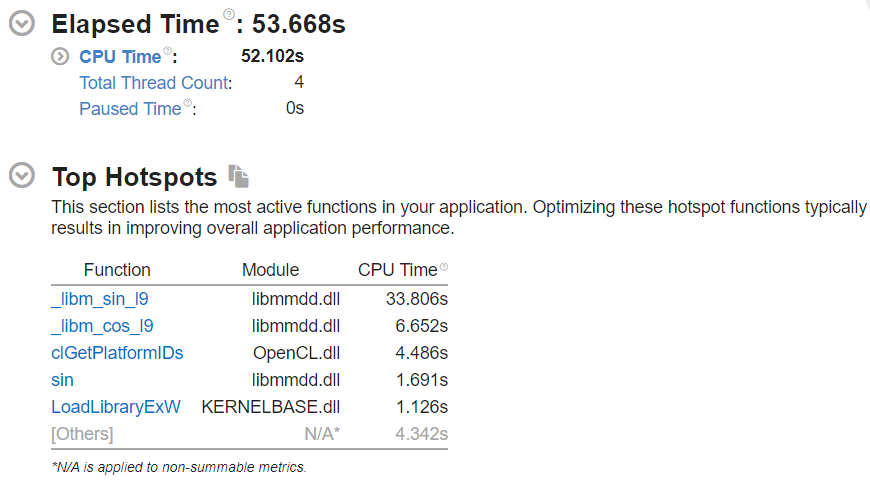
xi∈ [-500, 500], xopt = 420.97, yopt = -418.9829.

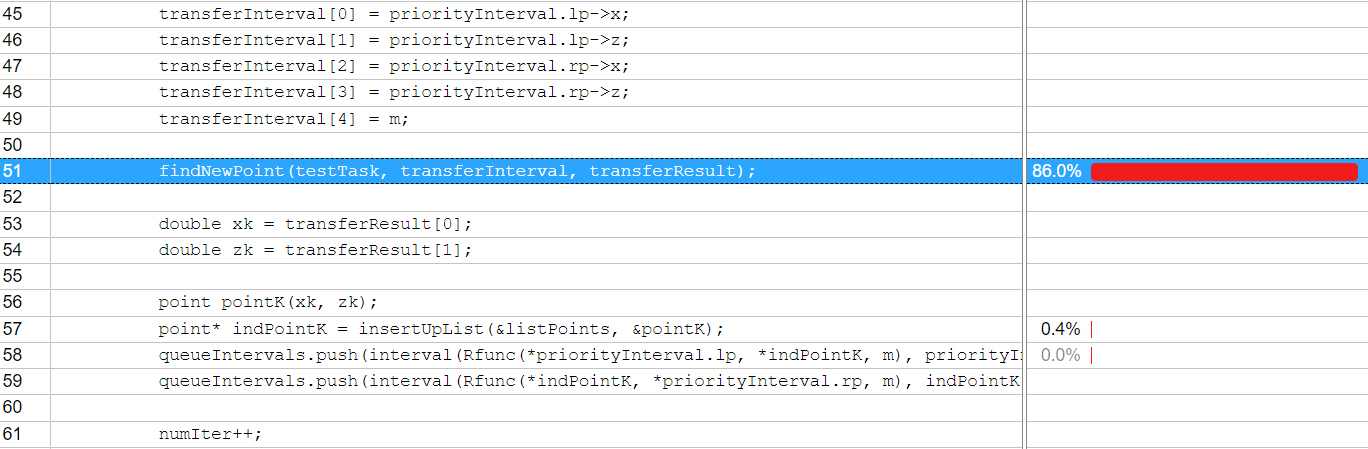
В функции используется задержка, чтобы представить её выполнение как трудоёмкий процесс.

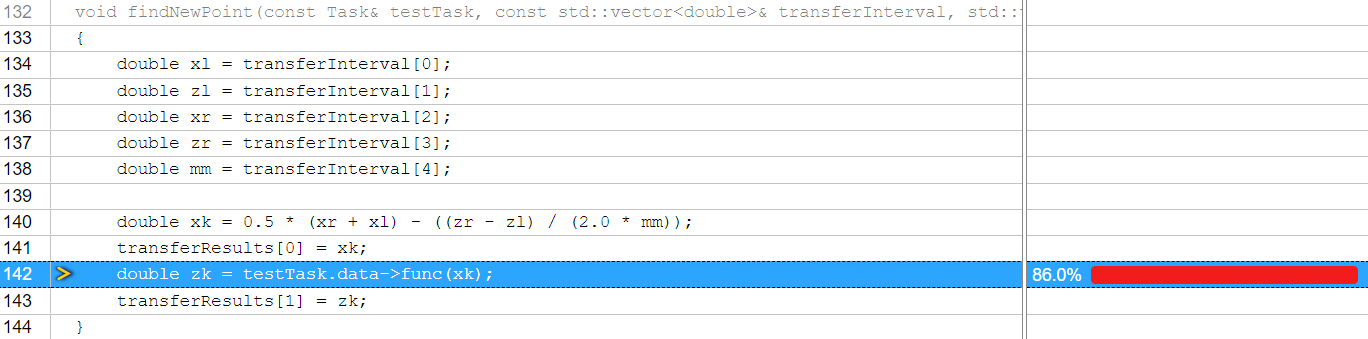
Характеристики тестовой машины:

* OS: Windows 10
* CPU: Intel Core i5-7200U 2,7 GHz
* RAM: 8Gb
* Конфигурация сборки: x64 Debug

Проведем hotspot анализ производительности, используя Intel® VTune™ Profiler.



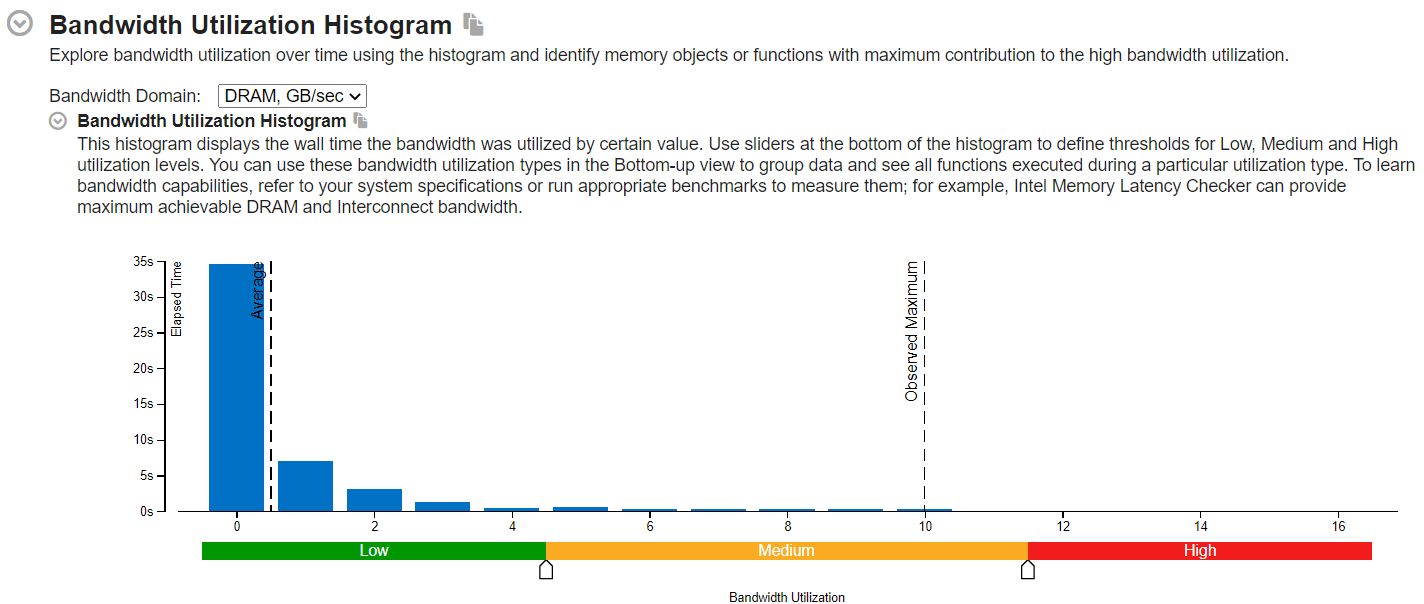




Как и ожидалось, основное время занимает вычисление тестовой функции.

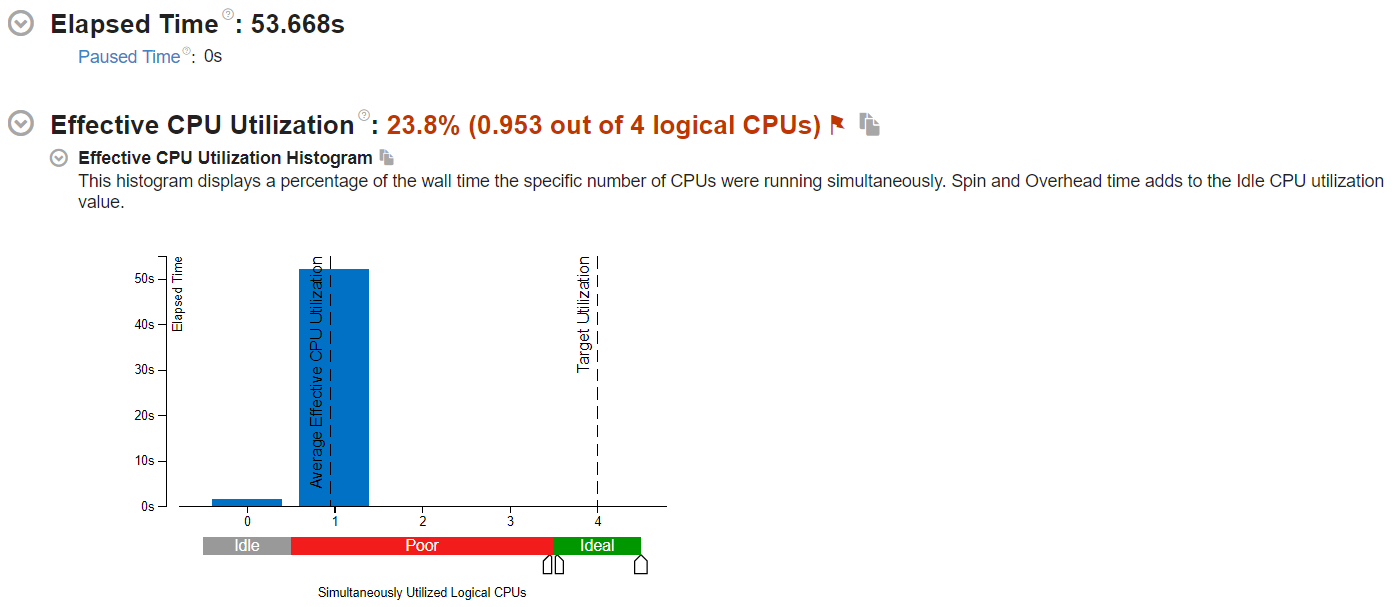
Запустим Memory Access анализ.





Как видно из анализа в целом программа не сталкивается с ограничениями по пропускной способности памяти.

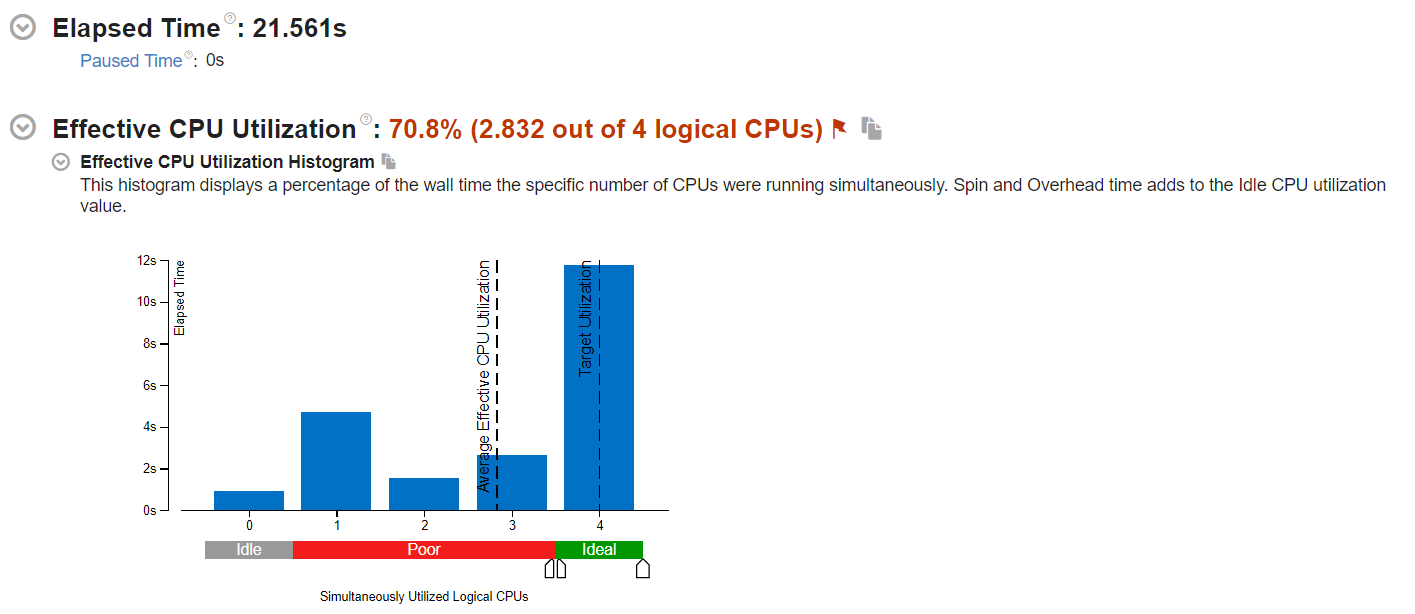
Проведём анализ параллельности кода.



Программа выполняется последовательно, не используя все вычислительные ядра процессора.

# **Процесс оптимизации**

Для решения проблемы неэффективного использования ядер процессора используем библиотеку OpenMP. Распараллелим функцию отыскания новой точки и проведем повторный анализ с помощью Vtune.



Наблюдается существенный прирост производительности.

# **Заключение**

Проведя анализ производительности исходной программы, была выявлена проблема с неэффективным использованием ядер процессора. Используя технологию OpenMP удалось ускорить выполнение программы практически в 2.5 раза.

# **Приложение. Код программы**

Исходный код программы доступен в репозитории <https://github.com/J-win/TestProblemOptimisation>.

## **Problem.h**

#pragma once

#include <vector>

#include <cmath>

#include <iostream>

#include <string>

struct Problem

{

double a;

double b;

double xopt;

double zopt;

double r;

const double pi = 3.14159265358979323846;

const double e = 2.71828182845904523536;

double summ(double sum);

virtual double func(const double x) = 0;

virtual std::string getNameFunc() = 0;

void getInfo();

};

struct SchwefelProblem : public Problem

{

SchwefelProblem(const double r\_ = 2.0);

double func(const double x) override;

std::string getNameFunc() override;

};

struct Task

{

Problem\* data;

Task();

~Task();

};

## **Problem.cpp**

#include "Problem.h"

double Problem::summ(double sum) {

for (int i = 1; i <= 100000; i++) {

sum += sin(sin(sin(i))) \* sin(sin(sin(i))) + cos(sin(sin(i))) \* cos(sin(sin(i)));

}

sum -= 100000;

return sum;

}

void Problem::getInfo()

{

std::cout << getNameFunc() << std::endl;

std::cout << "Optimum arg min f = " << xopt << std::endl;

std::cout << "Optimum min f = " << zopt << std::endl;

}

SchwefelProblem::SchwefelProblem(const double r\_)

{

a = -500.0;

b = 500.0;

xopt = 420.97;

zopt = -418.9829;

r = r\_;

}

double SchwefelProblem::func(const double x)

{

return -1.0 \* x \* sin(sqrt(fabs(x))) + summ(0.0);

}

std::string SchwefelProblem::getNameFunc()

{

return "Schwefel problem dimension one";

}

Task::Task()

{

data = new SchwefelProblem;

}

Task::~Task()

{

delete[] data;

}

## **SupportStructures.h**

#pragma once

#include <list>

#include <queue>

#include <vector>

#include "Problem.h"

struct point

{

double x;

double z;

point(const double x\_, const double z\_) : x(x\_), z(z\_) {}

point(const point& p) : x(p.x), z(p.z) {}

};

struct interval

{

double R;

point\* lp;

point\* rp;

interval(const double R\_ = 0.0, point\* lp\_ = nullptr, point\* rp\_ = nullptr) : R(R\_), lp(lp\_), rp(rp\_) {}

interval(const interval& i)

{

R = i.R;

lp = i.lp;

rp = i.rp;

}

interval& operator=(const interval& i)

{

R = i.R;

lp = i.lp;

rp = i.rp;

return \*this;

}

};

bool operator<(const interval& i1, const interval& i2)

{

return (i1.R < i2.R) ? true : false;

}

double Rfunc(const point& lp\_, const point& rp\_, const double m)

{

double dx = rp\_.x - lp\_.x;

double dz = rp\_.z - lp\_.z;

return (m \* dx + dz \* dz / (m \* dx) - 2.0 \* (rp\_.z + lp\_.z));

}

point\* insertUpList(std::list<point>\* p, point\* xk)

{

std::list<point>::iterator itl, itr;

itl = itr = (\*p).begin();

while ((itr != (\*p).end()) && (itr->x < (\*xk).x))

{

itl = itr;

itr++;

}

(\*p).insert(itr, (\*xk));

itl++;

return &(\*itl);

}

double funcFindM(std::list<point>& listPoints, const double r)

{

std::list<point>::iterator itl, itr;

double mm = 0.0;

double m;

itr = itl = listPoints.begin();

itr++;

while (itr != listPoints.end())

{

double max = fabs((itr->z - itl->z) / (itr->x - itl->x));

if (mm < max)

{

mm = max;

}

itr++;

itl++;

}

if (mm > 0.0)

{

m = r \* mm;

}

else

{

m = 1.0;

}

return m;

}

std::pair<double, double> funcFindMinInList(std::list<point>& listPoints)

{

std::list<point>::iterator itl;

itl = listPoints.begin();

double minf = itl->z;

double minx = itl->x;

itl++;

while (itl != listPoints.end())

{

if (minf > itl->z)

{

minf = itl->z;

minx = itl->x;

}

itl++;

}

return std::pair<double, double>(minx, minf);

}

void refillingQueue(std::list<point>& listPoints, std::priority\_queue<interval>& queueIntervals, const double m)

{

std::list<point>::iterator itl, itr;

while (!queueIntervals.empty())

{

queueIntervals.pop();

}

itr = itl = listPoints.begin();

itr++;

while (itr != listPoints.end())

{

queueIntervals.push(interval(Rfunc(\*itl, \*itr, m), &(\*itl), &(\*itr)));

itl++;

itr++;

}

}

void findNewPoint(const Task& testTask, const std::vector<double>& transferInterval, std::vector<double>& transferResults)

{

double xl = transferInterval[0];

double zl = transferInterval[1];

double xr = transferInterval[2];

double zr = transferInterval[3];

double mm = transferInterval[4];

double xk = 0.5 \* (xr + xl) - ((zr - zl) / (2.0 \* mm));

transferResults[0] = xk;

double zk = testTask.data->func(xk);

transferResults[1] = zk;

}

## **App.cpp**

#include <queue>

#include <cstdlib>

#include <chrono>

#include <omp.h>

#include "Problem.h"

#include "SupportStructures.h"

int AGSLinearVersion(int argc, char\* argv[])

{

Task testTask;

int maxNumIter = 2000;

double eps = 0.0000001;

int end = 1;

std::list<point> listPoints;

std::priority\_queue<interval> queueIntervals;

double x;

std::vector<double> transferInterval(5);

std::vector<double> transferResult(2);

double m = -1.0;

int numIter = 0;

auto st = std::chrono::steady\_clock::now();

x = testTask.data->a;

listPoints.push\_back(point(x, testTask.data->func(x)));

x = testTask.data->b;

listPoints.push\_back(point(x, testTask.data->func(x)));

do

{

double mold = m;

m = funcFindM(listPoints, testTask.data->r);

if (mold != m)

{

refillingQueue(listPoints, queueIntervals, m);

}

interval priorityInterval = queueIntervals.top();

queueIntervals.pop();

transferInterval[0] = priorityInterval.lp->x;

transferInterval[1] = priorityInterval.lp->z;

transferInterval[2] = priorityInterval.rp->x;

transferInterval[3] = priorityInterval.rp->z;

transferInterval[4] = m;

findNewPoint(testTask, transferInterval, transferResult);

double xk = transferResult[0];

double zk = transferResult[1];

point pointK(xk, zk);

point\* indPointK = insertUpList(&listPoints, &pointK);

queueIntervals.push(interval(Rfunc(\*priorityInterval.lp, \*indPointK, m), priorityInterval.lp, indPointK));

queueIntervals.push(interval(Rfunc(\*indPointK, \*priorityInterval.rp, m), indPointK, priorityInterval.rp));

numIter++;

end = 1;

if (priorityInterval.rp->x - priorityInterval.lp->x <= eps)

{

end = 0;

}

if (numIter >= maxNumIter)

{

end = 0;

}

} while (end != 0);

std::pair<double, double> optimum = funcFindMinInList(listPoints);

auto fi = std::chrono::steady\_clock::now();

std::cout << "Arg min f = " << optimum.first << std::endl;

std::cout << "Min f = " << optimum.second << std::endl;

std::cout << "Number iterations = " << numIter << std::endl;

std::cout << "Linear time work = " << std::chrono::duration\_cast<std::chrono::milliseconds>(fi - st).count() / 1000.0 << std::endl;

std::cout << std::endl;

testTask.data->getInfo();

return 0;

}

int AGSParallelVersion(int argc, char\* argv[])

{

Task testTask;

int maxNumIter = 2000;

double eps = 0.0000001;

int numThreads = 4;

int end = 1;

std::list<point> listPoints;

std::priority\_queue<interval> queueIntervals;

double x;

std::vector<std::vector<double>> transferIntervals(numThreads, std::vector<double>(5));

std::vector<std::vector<double>> transferResults(numThreads, std::vector<double>(2));

std::vector<interval> priorityIntervals(numThreads);

double m = -1.0;

int numIter = 0;

auto st = std::chrono::steady\_clock::now();

double pr = (testTask.data->b - testTask.data->a) / numThreads;

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

x = testTask.data->a + pr \* i;

listPoints.push\_back(point(x, testTask.data->func(x)));

}

x = testTask.data->b;

listPoints.push\_back(point(x, testTask.data->func(x)));

do

{

double mold = m;

m = funcFindM(listPoints, testTask.data->r);

if (mold != m)

{

refillingQueue(listPoints, queueIntervals, m);

}

for (int numThread = 0; numThread < numThreads; numThread++)

{

priorityIntervals[numThread] = queueIntervals.top();

queueIntervals.pop();

transferIntervals[numThread][0] = priorityIntervals[numThread].lp->x;

transferIntervals[numThread][1] = priorityIntervals[numThread].lp->z;

transferIntervals[numThread][2] = priorityIntervals[numThread].rp->x;

transferIntervals[numThread][3] = priorityIntervals[numThread].rp->z;

transferIntervals[numThread][4] = m;

}

#pragma omp parallel num\_threads(numThreads)

{

int numThread = omp\_get\_thread\_num();

findNewPoint(testTask, transferIntervals[numThread], transferResults[numThread]);

}

for (int numThread = 0; numThread < numThreads; numThread++)

{

double xk = transferResults[numThread][0];

double zk = transferResults[numThread][1];

point pointK(xk, zk);

point\* indPointK = insertUpList(&listPoints, &pointK);

queueIntervals.push(interval(Rfunc(\*priorityIntervals[numThread].lp, \*indPointK, m), priorityIntervals[numThread].lp, indPointK));

queueIntervals.push(interval(Rfunc(\*indPointK, \*priorityIntervals[numThread].rp, m), indPointK, priorityIntervals[numThread].rp));

numIter++;

}

end = 1;

for (int numThread = 0; numThread < numThreads; numThread++)

{

if (priorityIntervals[numThread].rp->x - priorityIntervals[numThread].lp->x <= eps)

{

end = 0;

}

}

if (numIter >= maxNumIter)

{

end = 0;

}

} while (end != 0);

std::pair<double, double> optimum = funcFindMinInList(listPoints);

auto fi = std::chrono::steady\_clock::now();

std::cout << "Arg min f = " << optimum.first << std::endl;

std::cout << "Min f = " << optimum.second << std::endl;

std::cout << "Number iterations = " << numIter << std::endl;

std::cout << "Parallel time work = " << std::chrono::duration\_cast<std::chrono::milliseconds>(fi - st).count() / 1000.0 << std::endl;

std::cout << std::endl;

testTask.data->getInfo();

return 0;

}

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

{

return AGSParallelVersion(argc, argv);

}