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

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 x^* и значений φ^* абсолютного минимума действительной функции $\varphi(x)$ на отрезке [a,b] вещественной оси x:

$$\varphi^* = \varphi(x^*) = \min \varphi(x), \qquad x \in [a, b].$$

Вычислительная схема алгоритма:

0) Первые два испытания осуществляются в точках $x^0 = a$ и $x^1 = b$.
1) Перенумеровать нижним индексом точки $x^i, 0 \le i \le k$ в порядке возрастания значения координаты.
2) Найти $M = \max \left|\frac{z_i - z_{i-1}}{x_i - x_{i-1}}\right|$, где $1 \le i \le k$, $z_i = \varphi(x_i)$.
3) Положить $m = \begin{cases} 1, & M = 0 \\ r * M, & M > 0 \end{cases}$, где $r > 1$.
4) Для каждого интервала (x_{i-1}, x_i), $1 \le i \le k$ вычислить

$$R(i) = m * (x_i - x_{i-1}) + \frac{(z_i - z_{i-1})^2}{m * (x_i - x_{i-1})} - 2 * (z_i + z_{i-1}),$$

где $R -$ характеристика интервала.

5) Найти t, при котором $R(t) = \max\big(R(i)\big), 1 \le i \le k$.
6) Положить $x^{k+1} = \frac{x_t + x_{t-1}}{2} - \frac{z_t - z_{t-1}}{2 * m}$.

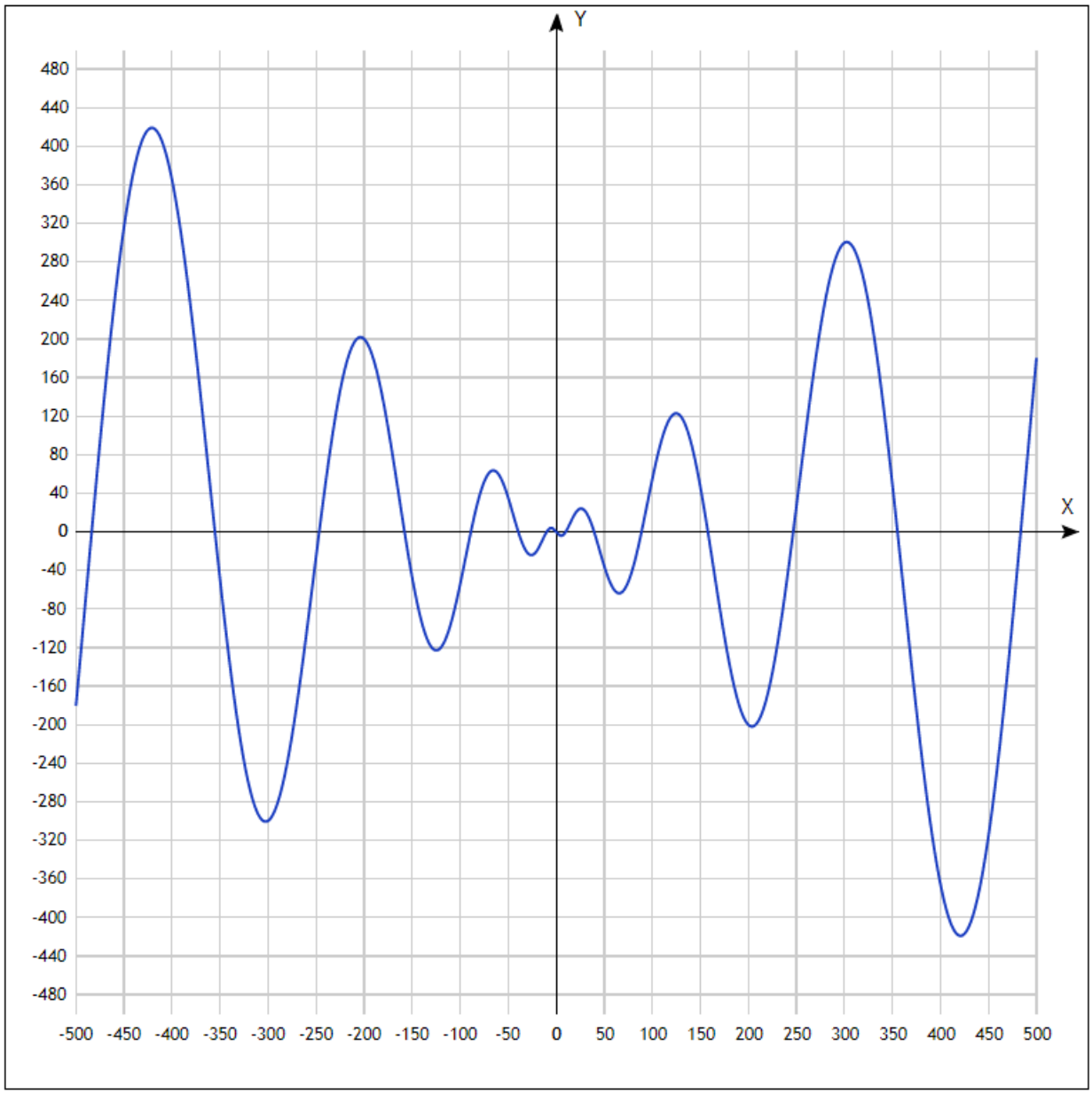
С помощью данного алгоритма добавляем точки в список найденных точек. Далее выполняем добавление точек, пока не достигнем требуемой точности ($x_t - x_{t-1} \le \varepsilon$), или пока не будет выполнено заданное число испытаний.

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.



$y(x) = -xsin(\sqrt[2]{|x|})$

Рис. 1. График функции Schwefel в одномерном случае

Для этой функции известен оптимум и границы его поиска:

$x_i \in [-500, 500]$, 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.

Elapsed Time: 53.668s

CPU Time:	52.102s
Total Thread Count:	4
Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time
_libm_sin_l9	libmmdd.dll	33.806s
_libm_cos_l9	libmmdd.dll	6.652s
clGetPlatformIDs	OpenCL.dll	4.486s
sin	libmmdd.dll	1.691s
LoadLibraryExW	KERNELBASE.dll	1.126s
[Others]	N/A*	4.342s

*N/A is applied to non-summable metrics.

```
45          transferInterval[0] = priorityInterval.lp->x;
46          transferInterval[1] = priorityInterval.lp->z;
47          transferInterval[2] = priorityInterval.rp->x;
48          transferInterval[3] = priorityInterval.rp->z;
49          transferInterval[4] = m;
50
51          findNewPoint(testTask, transferInterval, transferResult);          86.0%
52
53          double xk = transferResult[0];
54          double zk = transferResult[1];
55
56          point pointK(xk, zk);
57          point* indPointK = insertUpList(&listPoints, &pointK);             0.4%
58          queueIntervals.push(interval(Rfunc(*priorityInterval.lp, *indPointK, m), priorityI  0.0%
59          queueIntervals.push(interval(Rfunc(*indPointK, *priorityInterval.rp, m), indPointK
60
61          numIter++;
132 void findNewPoint(const Task& testTask, const std::vector<double>& transferInterval, std::
133 {
134     double xl = transferInterval[0];
135     double zl = transferInterval[1];
136     double xr = transferInterval[2];
137     double zr = transferInterval[3];
138     double mm = transferInterval[4];
139
140     double xk = 0.5 * (xr + xl) - ((zr - zl) / (2.0 * mm));
141     transferResults[0] = xk;
142 >   double zk = testTask.data->func(xk);                                   86.0%
143     transferResults[1] = zk;
144 }
```

Как и ожидалось, основное время занимает вычисление тестовой функции.

Запустим Memory Access анализ.

Elapsed Time: 47.165s

CPU Time:	46.255s	
Memory Bound:	**18.0%**	**of Pipeline Slots**
Loads:	67,811,634,288	
Stores:	25,659,169,752	
LLC Miss Count:	0	
Average Latency (cycles):	8	
Total Thread Count:	6	
Paused Time:	0s	

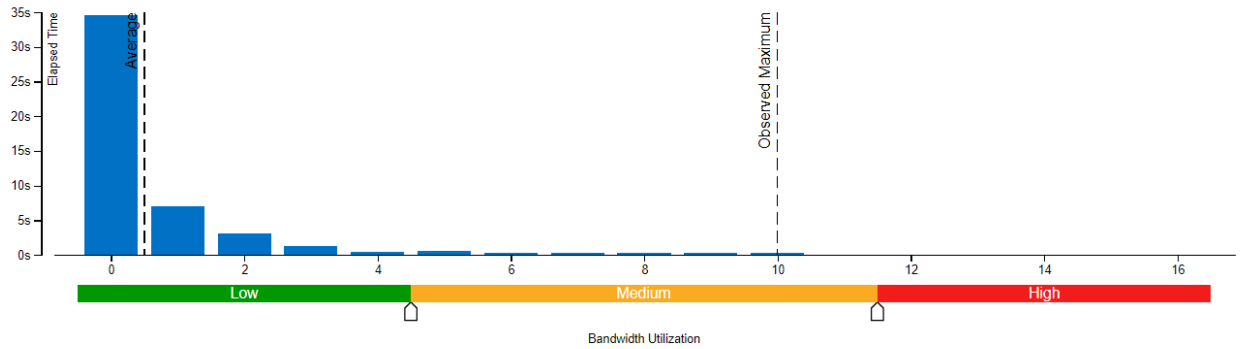
Bandwidth Utilization Histogram

Explore bandwidth utilization over time using the histogram and identify memory objects or functions with maximum contribution to the high bandwidth utilization.

Bandwidth Domain: DRAM, GB/sec

Bandwidth Utilization Histogram

This histogram displays the wall time the bandwidth was utilized by certain value. Use sliders at the bottom of the histogram to define thresholds for Low, Medium and High utilization levels. You can use these bandwidth utilization types in the Bottom-up view to group data and see all functions executed during a particular utilization type. To learn bandwidth capabilities, refer to your system specifications or run appropriate benchmarks to measure them; for example, Intel Memory Latency Checker can provide maximum achievable DRAM and Interconnect bandwidth.



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

Проведём анализ параллельности кода.

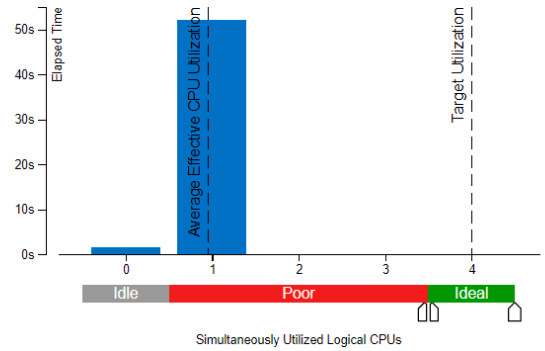
Elapsed Time: 53.668s

Paused Time: 0s

Effective CPU Utilization: 23.8% (0.953 out of 4 logical CPUs)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



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.

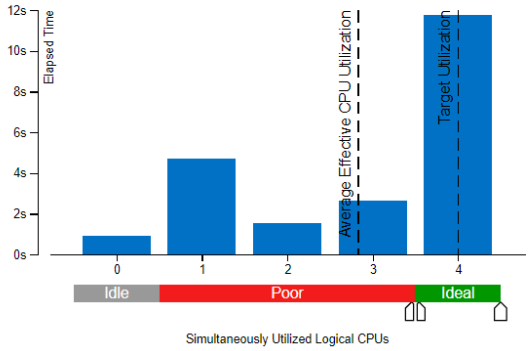
Elapsed Time: 21.561s

Paused Time: 0s

Effective CPU Utilization: 70.8% (2.832 out of 4 logical CPUs)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.



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);
}
```