Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского (ННГУ)

Институт информационных технологий, математики и механики

Кафедра математического обеспечения и суперкомпьютерных технологий

Курсовая работа

«Параллельные вычисления в задачах глобальной оптимизации»

Выполнил:
Студент группы 381503-3
Лалыкин Олег Вадимович

_____
Подпись

Научный руководитель:
Доцент кафедры МОСТ, к.т.н.
Сысоев Александр Владимирович

_____
Подпись

Нижний Новгород
2018

Содержание

Введение

Большое количество постановок технических или научных проблем можно сформулировать как задачу глобальной оптимизации. В связи с чем, алгоритмы глобальной оптимизации находят широкое применение в разнообразных областях науки и техники, везде, где необходимо получить наилучший результат целевой функции, оценивающей качество принимаемого решения.

К настоящему времени разработано большое количество алгоритмов и методов решения задачи многоэкстремальной оптимизации. Сегодня разработка методов глобальной оптимизации стимулируется развитием электронно-вычислительных средств и во многом связана с доступностью параллельных компьютерных систем высокой производительности. Появилось большое количество разнообразных подходов к распараллеливанию алгоритмов глобальной оптимизации.

Для одномерных функций существует множество стратегий многоэкстремальной оптимизации. Одним из эффективных методов одномерной глобальной оптимизации является информационный алгоритм Р. Г. Стронгина. Решение многомерной задачи может быть сведено к решению одномерной задачи путем редукции размерности с использованием кривых Пеано.

Постановка задачи

1. Изучение характеристически представимых методов решения одномерных задач глобальной оптимизации.
2. Изучение типовых схем распараллеливания характеристически представимых методов решения одномерных задач глобальной оптимизации.
3. Реализация некоторых типовых схем распараллеливания для решения одномерных задач глобальной оптимизации.
4. Изучение методов решения многомерных задач глобальной оптимизации.
5. Реализация решения двумерной задачи глобальной оптимизации.

В рамках данной работы требуется реализовать:

1. Последовательный и параллельный алгоритм решения одномерной задачи глобальной оптимизации Р. Г. Стронгина.
2. Последовательный и параллельный алгоритм решения двумерной задачи глобальной оптимизации с использованием схемы редукции размерности кривыми Пеано.
3. Тестирование для проверки корректности, используя функции Гришагина.

Описание алгоритмов

Алгоритм глобального поиска[1]

Рассмотрим одномерную задачу минимизации функции $\varphi(x)$ на отрезке [a, b].

Согласно алгоритму, два первых испытания проводятся на концах отрезка [a, b], то есть

$x_1 = a$, $x_2 = b$, далее вычисляются значения функции $z_1 = \varphi(a)$, $z_2 = \varphi(b)$, и количество проведенных испытаний κ полагается равным 2.

Пусть проведено $k \geq 2$ испытаний и получена информация

$$\omega = \omega_k = \{(x_i, z_i), 1 \leq i \leq k\}$$

Для выбора точки x_{k+1}нового испытания необходимо выполнить следующие действия.

1. Перенумеровать нижним индексом (начиная с нулевого значения) точки
   x_i, $1 \leq i \leq k$, в порядке возрастания:$a = x_0 < x_1 < \cdots < x_{k-1} = b$

2. Полагая $z_i = \varphi(x_i), 1 \leq i \leq k$, вычислить величины:

   $M = \max\limits_{1 \leq i \leq k-1} \left|\frac{z_i - z_{i-1}}{x_i - x_{i-1}}\right|$ и $m = \begin{cases} rM, & M > 0 \\ 1, & M = 0 \end{cases}$

   где $r > 1$ является заданным параметром метода.

3. Для каждого интервала $(x_{i-1}, x_i), 1 \leq i \leq k-1$ вычислить характеристику

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

4. Найти интервал (x_{t-1}, x_t), которому соответствует максимальная характеристика

   $$R(t) = \max\{R(i): 1 \leq i \leq k-1\}$$

5. Провести новое испытание в точке:

   $$x_{k+1} = \frac{1}{2}(x_t + x_{t-1}) - \frac{z_t - z_{t-1}}{2m}$$

6. Вычислить значение $z_{k+1} = \varphi(x_{k+1})$ и увеличить номер шага на единицу
   $k = k + 1$.

[1] Источник – литература [1]

Операции пунктов 1-5 описывают решающее правило АГП. Правило остановки задается в форме:

$$H_k(\Phi, \omega_k) = \begin{cases} 0, & x_t - x_{t-1} \le \varepsilon \\ 1, & x_t - x_{t-1} > \varepsilon \end{cases}$$

где $\varepsilon > 0$ – заданная точность поиска (по координате).

В качестве оценки экстремума выбирается пара: $e^k = (\varphi_k^*, x_k^*)$.

φ_k^* - минимальное вычисленное значение функции: $\varphi_k^* = \min\limits_{1 \le i \le k} \varphi(x_i)$.

x_k^* - координата этого значения: $x_k^* = \arg\min\limits_{1 \le i \le k} \varphi(x_i)$.

Адаптация к двумерной задаче [2]

Описанный алгоритм глобального поиска одномерной функции можно использовать для решения многомерной задачи.

Для этого используется способ редукции размерности отображением многомерной области поиска на одномерный интервал с помощью кривых Пеано.

Пусть дана задача глобальной оптимизации вида:

$$\phi^* = \phi(y^*) = \min\{\phi(y): y \in D, g_j(y) \le 0, 1 \le j \le m\},$$

где $D = \{y \in R^N: a_i \le y_i \le b_i, 1 \le i \le N\}$ - область в N-мерном пространстве

Целевая функция $\phi(y)$ удовлетворяют условию Липшица с соответствующими константами.

Используя кривые типа развертки Пеано $y(x)$, однозначно отображающие отрезок [0,1] на N-мерный гиперкуб D

$$D = \{y \in R^N: -2^{-1} \le y_i \le 2^{-1}, 1 \le i \le N\} = \{y(x): 0 \le x \le 1\},$$

исходную задачу можно редуцировать к следующей одномерной задаче:

$$\phi(y(x^*)) = \min\{\phi(y(x)): x \in [0,1], g_j(y(x)) \le 0, 1 \le j \le m\}.$$

Пусть пространство в которой определена целевая функция $\phi(y1, y2)$ является двумерным (N=2), тогда область на которую отображается отрезок [0,1] имеет два ограничения : $a \le y1 \le b$, $c \le y2 \le d$.

[2] Источник – литература [1]

Таким образом, одномерный АГП адаптируется для решения двумерной задачи.

Алгоритм решения имеет вид:

Решается одномерная задача на единичном отрезке [0,1], где целевая функция $\phi(y1, y2)$ определена на двумерной области с заданными ограничениями $a \le y1 \le b$, $c \le y2 \le d$. Проводится итерация в соответствии с АГП, где очередная точка испытания $x \in [0,1]$ отображается на двумерную область $a \le y1 \le b$, $c \le y2 \le d$. В результате отображения берутся соответствующие координаты ($y1$, $y2$) из двумерной области и используются для отыскания значения целевой функции $\phi(y1, y2)$ на текущей итерации. Полученное значение целевой двумерной функции используется для определения очередной точки последующего испытания.

Развертки Пеано[3]

Само построение развертки, отображающей единичный отрезок [0,1] на N-мерный гиперкуб D

$$D=\{y \in R^N : -2^{-1} \le y_i \le 2^{-1},\ 1 \le i \le N\} = \{y(x) : 0 \le x \le 1\}$$

сводится к следующим операциям:

1. Разбиение гиперкуба D с длиной ребра равной 1 на 2^N гиперкубов первого разбиения с длиной ребра равной ½.
2. Далее каждый гиперкуб первого разбиения снова делится на 2^N следующего. Процесс деления продолжается пока гиперкубы не станут нужного разбиения.
3. Теперь аналогично поступают с единичным отрезком [0,1] – его так же делят на 2^N равных и продолжают делить каждую часть пока не достигнут нужного уровня разбиения.
4. Фиксируют, что точка $y(x)$, лежащая в гиперкубе уровня разбиения n, однозначно соответствует точке x, лежащей на отрезке того же разбиения n.
5. Фиксируют, что номера центров гиперкубов те же, что и номера соответствующих отрезков (на одном уровне разбиения).
6. Получают, что отображение отрезка [0,1] в гиперкуб D означает следующее: образ любого подынтервала из [0,1] является линейным отрезком, соединяющим центры-узлы гиперкуба.

[3] Источник – литература [1]

В итоге строится кусочно-линейная кривая, которая соединяет центры-узлы гиперкуба. Такая кривая является численным приближением к кривой Пеано с заданной точностью, которая зависит от плотности развертки.

Пример кусочно-линейной развертки для двумерной функции:

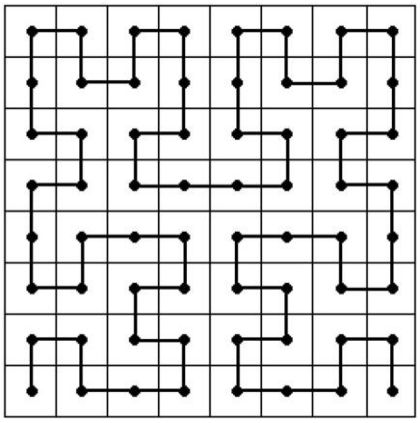

Рисунок 1 Развертка размерности 2 и плотности 3

Схемы распараллеливания задачи

Распараллеливание по области поиска

Алгоритм заключается в распределении исходного отрезка на независимые участки по потокам и выполнении последовательного АГП в каждом.

Основные действия схемы:

1. Исходный отрезок [a, b] разбивается на отдельные участки одинаковой длины в количестве равном числу выделенных потоков.
2. Каждый участок присваивается соответствующему потоку.
3. Каждый поток независимо от остальных выполняет последовательный АГП на своём участке.
4. Когда критерий остановки алгоритма выполнится во всех потоках, тогда происходит запись результатов.
5. Среди полученных результатов выбирается решение с наименьшим полученным значением целевой функции, он и считается решением всей задачи.

Распараллеливание по характеристикам

Алгоритм заключается в параллельном нахождении сразу нескольких характеристик $R(i)$, из которых в выбирается максимальная.

Основные действия схемы:

1. Параллельный поиск оценок константы Липшица

$$m = \begin{cases} rm, & M > 0 \\ 1, & M = 0 \end{cases}, \text{где } M = \max_{1 \le i \le k} \left| \frac{z_i - z_{i-1}}{x_i - x_{i-1}} \right|$$

2. Параллельно вычисляются значения характеристик и для каждого потока запоминается индекс и значение максимальной характеристики

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

3. Проводится новое испытание в точке x^{k+1}, где t – индекс максимальной характеристики

$$x^{k+1} = \frac{1}{2}(x_t + x_{t-1}) - \frac{(z_t - z_{t-1})}{2m}$$

Программная реализация

Структура программы

Evolvent.h – содержит заимствованный метод GetImage, реализованный в проекте Globalizer.

Grishagin_function.h – содержит методы, используемые для тестирования алгоритма на двумерных функциях Гришагина.

Search.h – содержит все реализованные в данной работе алгоритмы.

Описание класса “Search”

```
private:
	double left; // левая граница одномерной задачи
	double right; // правая граница одномерной задачи
	double* Left; // ограничение области слева двумерной задачи
	double* Right; // ограничение области справа двумерной задачи
	double r; // параметр метода
	int procs;//число потоков

public:
	Search(const double _left, const double _right, const double _r, const int _procs)//конструктор для одномерной задачи


	Search(const double *_left, const double *_right, const double _r, const int _procs)//конструктор для двумерной задачи

	double Func(const double &_x)//Одномерные функции

	double Func(const double *_y, vagrish::GrishaginFunction *func) //Двумерные функции Гришагина

	PointerOneDim Serial_searchMin(const double _left, const double _right, const int _N_max, const double _Eps);//Последовательный поиск одномерной задачи

	PointerOneDim Simple_Par_searchMin(const double _left, const double _right, const int _N_max, const double _Eps, const int _threads);//ПП по отрезкам одномерной задачи

	PointerOneDim Ch_SearchMin(const double _Epsilon, const int _Steps, const int threads);//ПП по характеристикам одномерной задачи

	PointerTwoDim Two_Dim_Search(const double _Epsilon, const int _Steps, const int threads, vagrish::GrishaginFunction *func);//Решатель двумерной задачи

	double R(const double &_m_small, const double &_z_curr, const double &_z_prev, const double &_x_curr, const double &_x_prev)//Характеристика

	double M(const double &_z_curr, const double &_z_prev, const double &_x_curr, const double &_x_prev)//Параметр для оценки константы Липшица

	double New_x(double &x_right, double &x_left, double &z_right, double &z_left, double &m_small)//Новая точка испытания

	void Lipschitz(double &M_big, double &m_small)//Оценка константы Липшица
```

Описание структур данных

`double left` - левая граница отрезка для одномерной задачи
`double right` - правая граница отрезка для одномерной задачи
`double* Left` - левые ограничения области для двумерной задачи
`double* Right` - правые ограничения области для двумерной задачи
`double r` - параметр метода
`int procs` - количество потоков

`struct PointerOneDim` – Структура для сбора результатов в одномерной задаче.
`struct PointerTwoDim`– Структура для сбора результатов в двумерной задаче.
`struct GroupOneDim` – Структура для организации параллельной реализации одномерной задачи. Распараллеливание по характеристикам.
`struct borders` – Структура для организации параллельной реализации одномерной задачи. Распараллеливание по области поиска.
`struct GroupTwoDim` – Структура для организации решения двумерной задачи.

Описание методов

`Search(const double _left, const double _right, const double _r, const int _procs)` - конструктор для одномерной задачи, инициализирует указанные входные параметры.

`Search(const double *_left, const double *_right, const double _r, const int _procs)` - конструктор для двумерной задачи, инициализирует указанные входные параметры.

`double Func(const double &_x)` – Одномерные целевые функции, зависящие от входного параметра `_x`

`double Func(const double *_y, vagrish::GrishaginFunction *func)` – Двумерные целевые функции Гришагина, зависящие от входного параметра `*_y`. Экземпляр `*func` передается для вызова соответствующего метода подсчёта значения целевой сгенерированной функции Гришагина.

`PointerOneDim Serial_searchMin(const double _left, const double _right, const int _N_max, const double _Eps)` – Последовательное решение одномерной задачи с заданными ограничениями.

`PointerOneDim Simple_Par_searchMin(const double _left, const double _right, const int _N_max, const double _Eps, const int _threads)` – Параллельное решение одномерной задачи с заданными ограничениями. Распараллеливание по области поиска.

`PointerOneDim Ch_SearchMin(const double _Epsilon, const int _Steps, const int threads)` – Параллельное решение одномерной задачи с заданными ограничениями. Распараллеливание по характеристикам.

`PointerTwoDim Two_Dim_Search(const double _Epsilon, const int _Steps, const int threads, vagrish::GrishaginFunction *func)` – Решение двумерной задачи с заданными ограничениями для указанной функции Гришагина.

`double R(const double &_m_small, const double &_z_curr, const double &_z_prev, const double &_x_curr, const double &_x_prev)` – Подсчёт характеристики.

`double M(const double &_z_curr, const double &_z_prev, const double &_x_curr, const double &_x_prev)` – Подсчёт параметра M для оценки константы Липшица.

`double New_x(double &x_right, double &x_left, double &z_right, double &z_left, double &m_small)` – Подсчёт следующей точки испытания.

`void Lipschitz(double &M_big, double &m_small)` – Подсчёт оценки константы Липшица.

Результаты

Тестирование для одномерной задачи

Проведем поиск глобального минимума для одномерной функции с одинаковыми параметрами при обычном критерии остановки.

Первый эксперимент проводится для последовательной версии алгоритма.

Пример решения одномерной задачи в последовательной версии

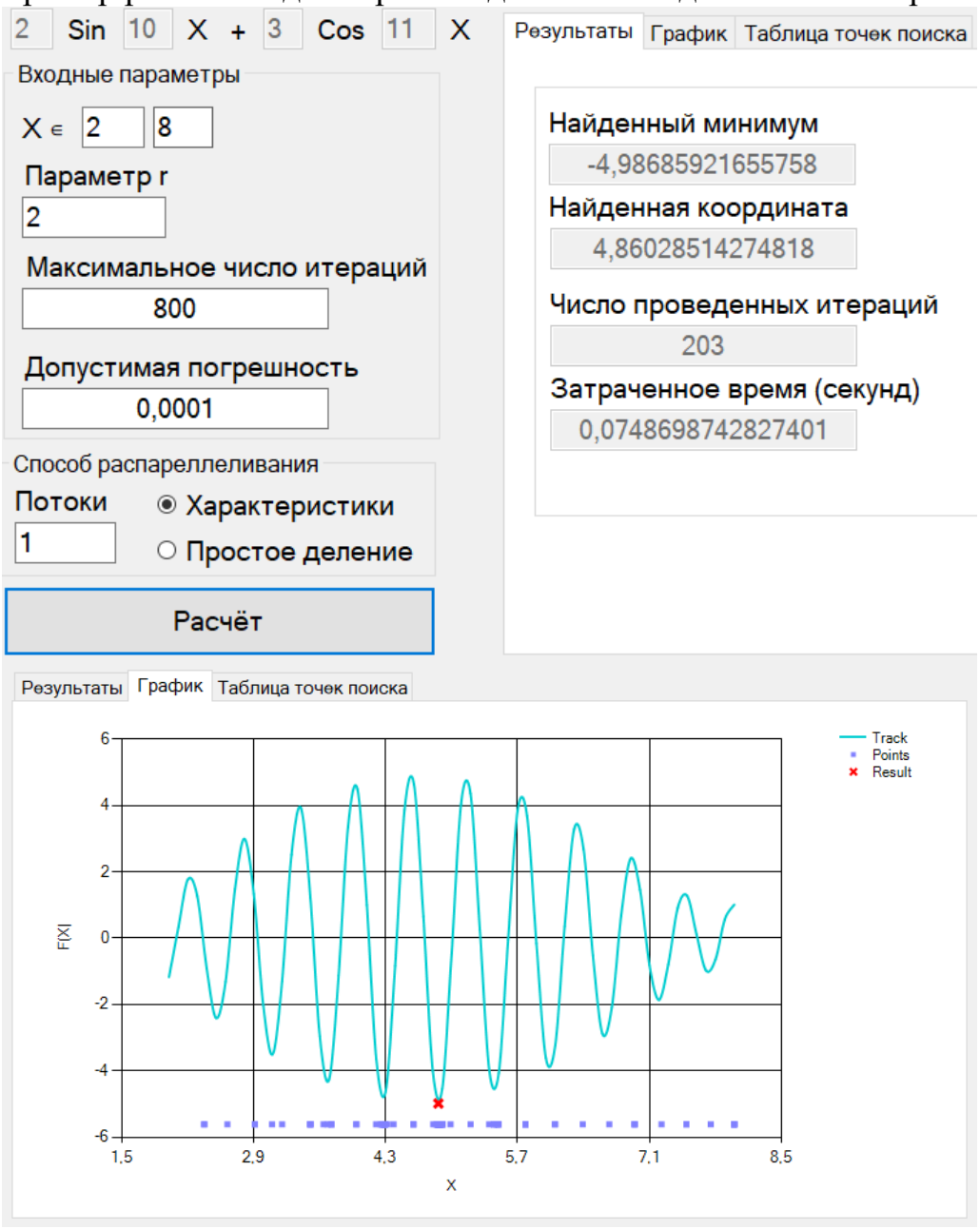

Рисунок 2 Результаты последовательной версии

Как видно по графику, решение можно назвать верным, так как указанная точка результата действительно находится близко к глобальному минимуму.

Следующий эксперимент проводится на четырёх потоках способом распараллеливания по характеристикам.

Пример распараллеливания по характеристикам

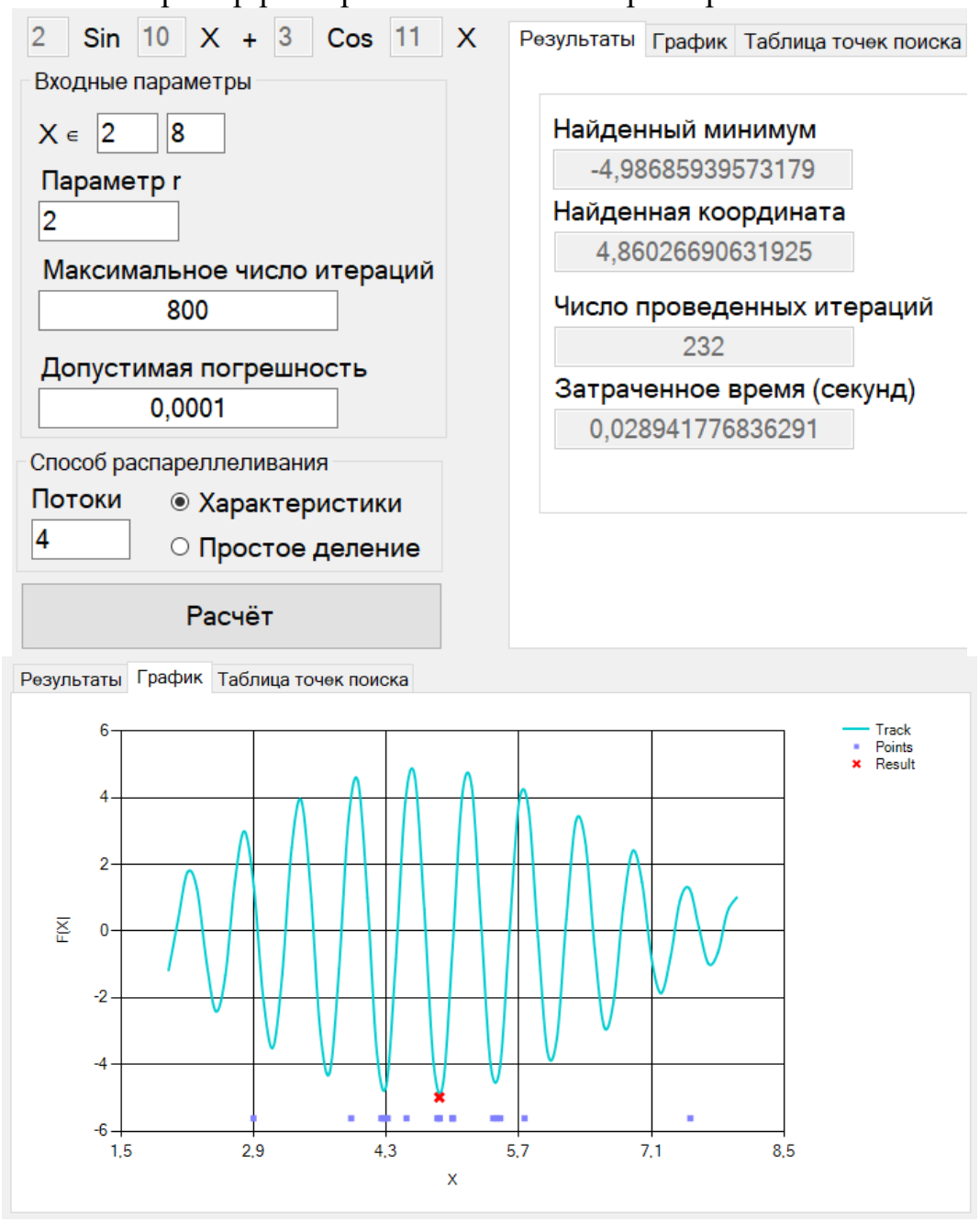

Рисунок 3 Результаты распараллеливания по характеристикам

Решение так же можно назвать верным. В сравнении с последовательной версией точки итерации менее разбросаны, это означает, что решение быстрее стекается к экстремуму. По затраченному времени можно так же наблюдать ускорение в работе.

Последний эксперимент так же проводится на четырёх потоках, но способ распараллеливания теперь ведется по области поиска.

Пример распараллеливания по области поиска

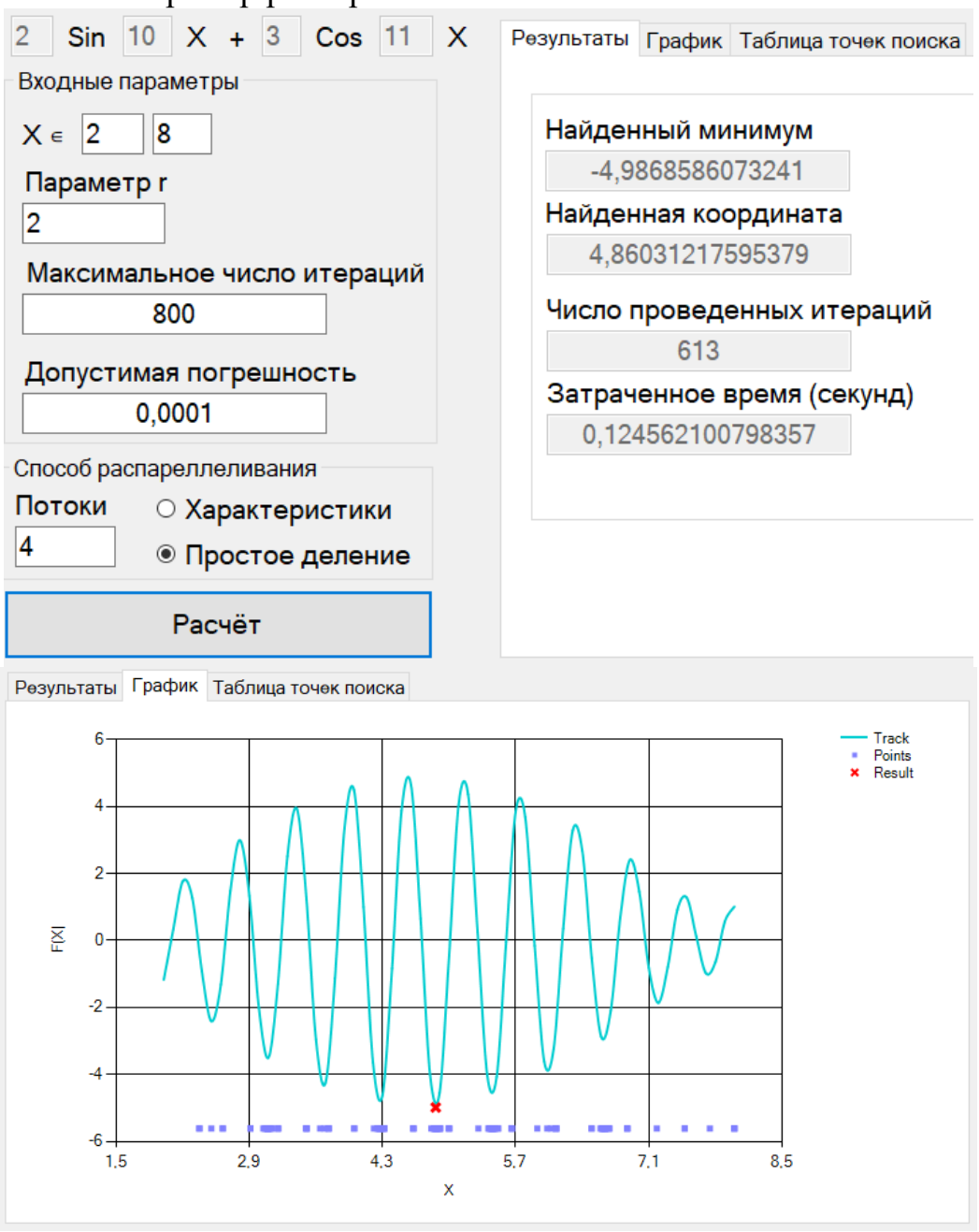

Рисунок 4 Результаты распараллеливания по области поиска

Здесь ситуация противоположна предыдущей. Количество точек испытаний резко возросло, так как потоки работают независимо и подавляющее число испытаний проводится впустую. Вследствие этого так же резко возросло время работы.

Этот способ распараллеливания крайне неэффективен, так как получается, что в нужной области работает только один поток и достигается условие остановки только им одним.

Проведем поиск глобального минимума при повышенном критерии остановки, чтобы лучше оценить ускорение при распараллеливании.

Проведем эксперимент в последовательной версии и распараллеленной по характеристикам.
Рассматривать распараллеливание по области поиска не имеет смысла, так как уже установлена его неэффективность.

Пример работы последовательной и параллельной версии при повышенном критерии остановки

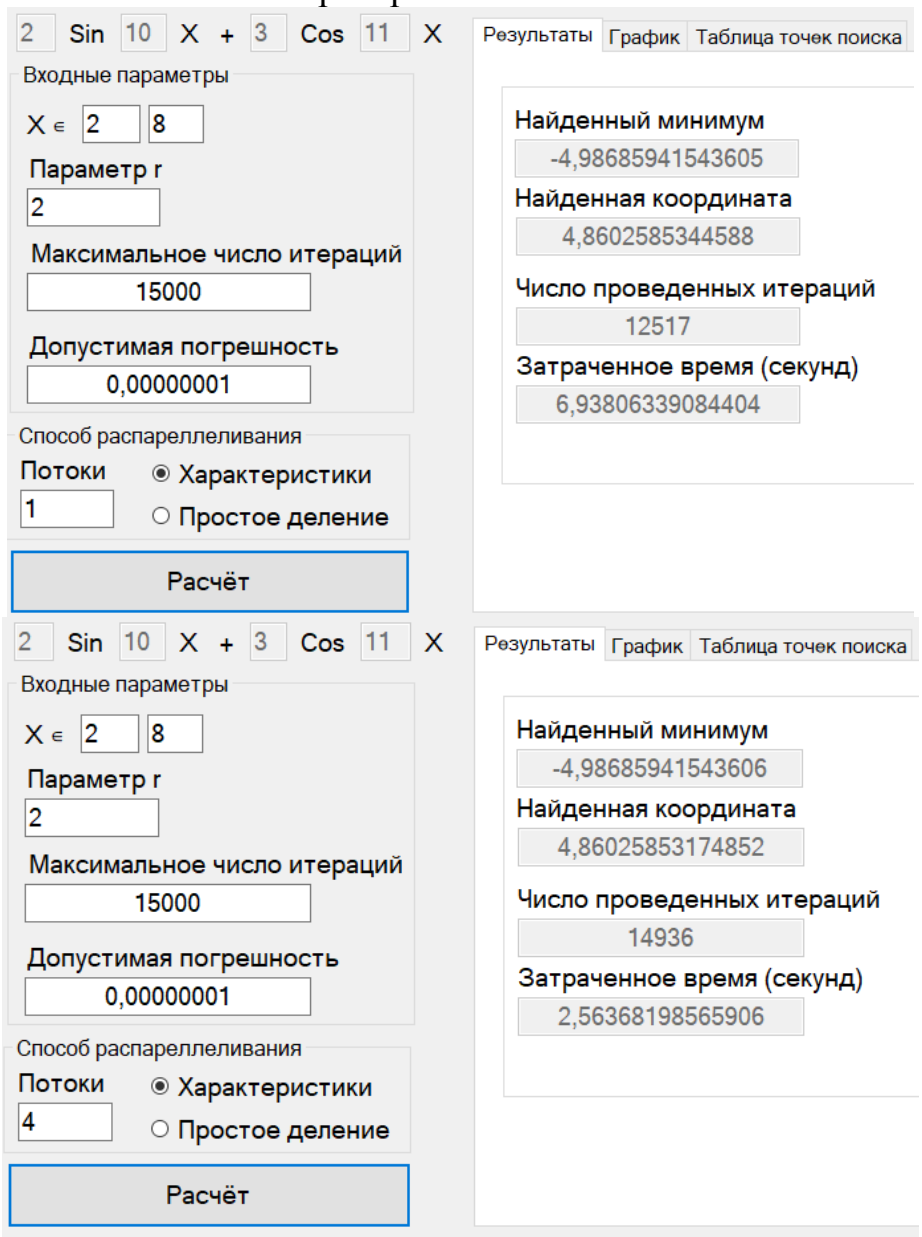

Рисунок 5 Сравнение при длительной работе

Действительно, в данном случае скорость работы параллельной версии в 2,7 раза выше последовательной из чего можно заключить, что параллельная версия достаточно эффективна.

Тестирование для двумерной задачи

Тестирование метода для функций Гришагина заключается в следующих действиях:

1. Выбор номера функции для генератора,
2. Получение входных параметров для функции – линейные ограничения на аргументы,
3. Проведение поиска глобального минимума для сгенерированной функции,
4. Сравнение полученного результата с известным-истинным,
5. Вынос результата – если разность между решениями не превышает заданную допустимую погрешность, то эксперимент успешен, иначе – провал.

Тестирование на 100 функциях Гришагина показало 100% правильность при следующих входных параметрах:

- Плотность развертки m = 20,
- Параметр метода r = 1.5,
- Критерий остановки по точности eps = 0.00001,
- Критерий остановки по числу итераций n = 1000,
- Число потоков - 4, распараллеливание по характеристикам,
- Условие корректности – погрешность не выше 0,1.

Пример тестирования для допустимой погрешности 0,01

```
GSA   : |Z :-10.715 |X: 0.203 |Y: 0.319
Truth: |Z :-10.739 |X: 0.198 |Y: 0.318 => Correct

GSA   : |Z :-9.059 |X: 0.877 |Y: 0.653
Truth: |Z :-9.06 |X: 0.876 |Y: 0.653 => Correct

GSA   : |Z :-10.745 |X: 0.229 |Y: 0.335
Truth: |Z :-10.749 |X: 0.23 |Y: 0.336 => Correct

GSA   : |Z :-8.455 |X: 0.17 |Y: 0.016
Truth: |Z :-8.455 |X: 0.169 |Y: 0.016 => Correct

GSA   : |Z :-11.295 |X: 0.761 |Y: 0.906
Truth: |Z :-11.296 |X: 0.76 |Y: 0.906 => Correct

GSA   : |Z :-10.465 |X: 0.703 |Y: 0.305
Truth: |Z :-10.476 |X: 0.703 |Y: 0.308 => Correct

GSA   : |Z :-9.284 |X: 0.363 |Y: 0.283
Truth: |Z :-9.286 |X: 0.365 |Y: 0.282 => Correct

GSA   : |Z :-9.582 |X: 0.314 |Y: 0.648
Truth: |Z :-9.593 |X: 0.314 |Y: 0.651 => Correct

GSA   : |Z :-10.802 |X: 0.236 |Y: 0.378
Truth: |Z :-10.821 |X: 0.238 |Y: 0.374 => Correct

GSA   : |Z :-9.352 |X: 0.585 |Y: 0.508
Truth: |Z :-9.347 |X: 0.586 |Y: 0.509 => Correct

GSA   : |Z :-9.775 |X: 0.003 |Y: 0.004
Truth: |Z :-9.804 |X: 0 |Y: 0 => Correct

GSA   : |Z :-10.8 |X: 0.386 |Y: 0.996
Truth: |Z :-10.862 |X: 0.383 |Y: 1 => Correct
```

Рисунок 6 Результаты теста из 12 функций

Заключение

В результате выполнения данной работы был разработан программный комплекс, реализующий поиск оптимума для одномерной и двумерной задач.

Проведя множество экспериментов, выполнив сравнение с заявленным оптимумом и найденным, можно судить о корректности и эффективности реализованных алгоритмов.

Литература

1. Р. Г. Стронгин, В. П. Гергель, В. А. Гришагин, К. А. Баркалов «Параллельные вычисления в задачах глобальной оптимизации»
2. Гергель В.П., Стронгин Р.Г Основы параллельных вычислений для многопроцессорных вычислительных систем Н. Новгород: Изд-во ННГУ, 2001

Приложение

Search.h

```
#ifndef __SEARCH_H__
#define __SEARCH_H__

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <queue>
#include <list>
#include <omp.h>

#include "grishagin\include\grishagin_function.hpp"
#include "evolvent.h"

struct PointerOneDim//структура для сбора результатов одномерной задачи
{
	double x, z;
	int steps;
	std::vector<double> X;
	double time;
};

struct PointerTwoDim//структура для сбора результатов двумерной задачи
{
	double x, y, z;
	int steps;
	std::vector<double> X;
	std::vector<double> Y;
	double time;
};

struct GroupOneDim//рабочая структура для ПП одномерной задачи по характеристикам
{
	double R; // оценка константы Липшица
	double M; // параметр оценки для интервала
	double x_left; // координата слева
	double x_right; // координата справа
	double z_left; // значение слева
	double z_right; // значение справа
};

struct borders//рабочая структура для ПП по отрезкам
{
	double left;
	double right;
};

struct GroupTwoDim//рабочая структура для двумерной задачи
{
	double R; // оценка константы Липшица
	double M; // параметр оценки для интервала

	//интервальные точки на единичном отрезке
	double x_new_left;
	double x_new_right;

	double x_left; // x координата слева
	double x_right; // x координата справа
	double y_left; // y координата слева
	double y_right; // y координата справа
	double z_left; // значение слева
```

```
	double z_right; // значение справа

};

class Search
{
private:
	double left; // левая граница одномерной задачи
	double right; // правая граница одномерной задачи

	double* Left; // ограничение области слева двумерной задачи
	double* Right; // ограничение области справа двумерной задачи

	double r; // параметр метода
	int procs;//число потоков

public:
	Search(const double _left, const double _right, const double _r, const int
_procs)//конструктор для одномерной задачи
	{
		left = _left;
		right = _right;
		r = _r;
		procs = _procs;
	}

	Search(const double *_left, const double *_right, const double _r, const int
_procs)//конструктор для двумерной задачи
	{
		Left = new double[2];
		Right = new double[2];
		for (int i = 0; i < 2; i++)
		{
			Left[i] = _left[i];
			Right[i] = _right[i];
		}
		r = _r;
		procs = _procs;
	}

	double Func(const double &_x)//Одномерные функции
	{
		return 2 * sin(10 * _x) + 3 * cos(11 * _x);
	}

	double Func(const double *_y, vagrish::GrishaginFunction *func) //Двумерные
функции Гришагина
	{
		return func->Calculate(_y);
	}

	PointerOneDim Serial_searchMin(const double _left, const double _right, const int
_N_max, const double _Eps);//Последовательный поиск одномерной задачи

	PointerOneDim Simple_Par_searchMin(const double _left, const double _right, const
int _N_max, const double _Eps, const int _threads);//ПП по отрезкам одномерной задачи

	PointerOneDim Ch_SearchMin(const double _Epsilon, const int _Steps, const int
threads);//ПП по характеристикам одномерной задачи

	PointerTwoDim Two_Dim_Search(const double _Epsilon, const int _Steps, const int
threads, vagrish::GrishaginFunction *func);//Решатель двумерной задачи
```

```
	double R(const double &_m_small, const double &_z_curr, const double &_z_prev,
const double &_x_curr, const double &_x_prev)//Характеристика
	{
		return _m_small * (_x_curr - _x_prev) + pow(_z_curr - _z_prev, 2) /
(_m_small*(_x_curr - _x_prev)) - 2 * (_z_curr + _z_prev);
	};

	double M(const double &_z_curr, const double &_z_prev, const double &_x_curr,
const double &_x_prev)//Параметр для оценки константы Липшица
	{
		return abs((_z_curr - _z_prev) / (_x_curr - _x_prev));
	};

	double New_x(double &x_right, double &x_left, double &z_right, double &z_left,
double &m_small)//Новая точка испытания
	{

		return  (x_right + x_left) / 2 - (z_right - z_left) / (2 * m_small);
	}
	void Lipschitz(double &M_big, double &m_small)//Оценка константы Липшица
	{
		if (M_big == 0)
		{
			m_small = 1;
		}
		else
		{
			m_small = r * M_big;
		}
	}

};


#endif
```

Two_Dim_Search.cpp

```
#include "Search.h"

PointerTwoDim Search::Two_Dim_Search(const double _Epsilon, const int _Steps, const int
threads, vagrish::GrishaginFunction *func)
{

	PointerTwoDim p;
	p.steps = 0;
	p.time = 0;

	int procs = threads;

	int localSteps = 0; // шаги для внутреннего счета

	double eps = _Epsilon;

	//текущие границы единичного отрезка
	double curr_left;
	double curr_right;

	// время работы
	double start, end;

	double *x_new = new double[procs];
```

```
	double *coordinates_y = new double[2];

	//константы для оценки конст Липшица
	double m_small = 1;
	double M_big;

	std::vector<GroupTwoDim> StartVec(procs);
	std::vector<GroupTwoDim> MainVec(procs * 2);
	std::list<GroupTwoDim> Local_group;

	size_t size_group; //// размер list<Group_2D> Local_group

	std::list<GroupTwoDim>::iterator Pointer;

	double  right_x_new, left_x_new;

	omp_set_dynamic(0);//среда выполнения не будет динамически настраивать количество потоков


	std::vector<TEvolvent*> evolve(procs);//вектор разверток для потоков (у каждого своя)

	for (int i = 0; i < procs; i++)//инициализация разверток каждому потоку
	{
		evolve[i] = new TEvolvent(2, 20);
		evolve[i]->SetBounds(Left, Right);
	}

	left_x_new = x_new[0] = 0;
	right_x_new = x_new[procs - 1] = 1;

	double startCooord[2];


	start = omp_get_wtime();

	omp_set_num_threads(procs);
	{
#pragma omp parallel for num_threads(procs)
		for (int i = 0; i < procs; i++)//распределение отрезка по потокам
		{

			StartVec[i].x_new_left = left_x_new + (i * (right_x_new - left_x_new) / procs);
			StartVec[i].x_new_right = StartVec[i].x_new_left + (right_x_new - left_x_new) / procs;

			evolve[i]->GetImage(StartVec[i].x_new_left, startCooord);

			StartVec[i].z_left = Func(startCooord, func);

			evolve[i]->GetImage(StartVec[i].x_new_right, startCooord);

			StartVec[i].z_right = Func(startCooord, func);
		}

		// подсчет M
#pragma omp parallel for num_threads(procs)
		for (int i = 0; i < procs; i++)
		{
			StartVec[i].M = M(StartVec[i].z_right, StartVec[i].z_left, StartVec[i].x_new_right, StartVec[i].x_new_left);
		}
```

```
            M_big = StartVec[0].M;

            for (int i = 1; i < procs; i++) // найдем макс из остальных
            {
                  if (M_big < StartVec[i].M)
                  {
                        M_big = StartVec[i].M;
                  }
            }
            Lipschitz(M_big, m_small);// оценка параметра m

             //подсчёт R
#pragma omp parallel for num_threads(procs)
            for (int i = 0; i < procs; i++)
            {
                  StartVec[i].R = R(m_small, StartVec[i].z_right, StartVec[i].z_left,
StartVec[i].x_new_right, StartVec[i].x_new_left);
            }
            for (register int i = 0; i < procs; i++)//вставка результатов в общий
список
            {
                  Local_group.push_back(StartVec[i]);
            }
            Local_group.sort();//сортировка по возрастанию

                                    //получение текущих границ
            curr_left = Local_group.back().x_new_left;
            curr_right = Local_group.back().x_new_right;

            localSteps++;

            while (true)//основной цикл
            {
                  Pointer = --Local_group.end();
                  for (int i = 0; i < procs * 2 - 2; i += 2) // хватаем proc_count R-ок
(самых больших)
                  {
                        MainVec[i] = *Pointer;
                        Local_group.erase(Pointer--);//остальные R-ки стираем (все
группы)
                  }
                  MainVec[procs * 2 - 2] = *Pointer;
                  Local_group.erase(Pointer);//.erase сотрет по указателю
                  size_group = Local_group.size();

                  // соответствующей  кучке больших R-ок находим соответствующие точки
испытаний
#pragma omp parallel for shared(x_new)  num_threads(procs)
                  for (int i = 0; i < procs; i++)
                  {
                        double coordinates_y[2];
                        x_new[i] = New_x(MainVec[i * 2].x_new_right, MainVec[i *
2].x_new_left, MainVec[i * 2].z_right, MainVec[i * 2].z_left, m_small);

                        evolve[i]->GetImage(x_new[i], coordinates_y);
                        MainVec[i * 2 + 1].x_left = coordinates_y[0];
                        MainVec[i * 2 + 1].y_left = coordinates_y[1];
                        MainVec[i * 2 + 1].z_left = Func(coordinates_y, func);
                  }

                  for (int i = 0; i < procs * 2; i += 2)//сохраняем найденные точки
                  {
                        MainVec[i + 1].x_right = MainVec[i].x_right;
```

```
                        MainVec[i + 1].y_right = MainVec[i].y_right;
                        MainVec[i + 1].x_new_right = MainVec[i].x_new_right;
                        MainVec[i + 1].z_right = MainVec[i].z_right;

                        MainVec[i + 1].x_new_left = MainVec[i].x_new_right = x_new[i /
2];

                        MainVec[i].x_right = MainVec[i + 1].x_left;
                        MainVec[i].y_right = MainVec[i + 1].y_left;
                        MainVec[i].z_right = MainVec[i + 1].z_left;
                }

                // считаем M
#pragma omp parallel for num_threads(procs)
                for (int i = 0; i < procs * 2; i++)
                {
                        MainVec[i].M = M(MainVec[i].z_right, MainVec[i].z_left,
MainVec[i].x_new_right, MainVec[i].x_new_left);
                }
                double M_max_array = MainVec[0].M;//найденная M одним потоком


        //поиск наибольшего M, который не соответствует текущей кучке (из M_max_all_M)
(а он может быть большим, чем все M из кучки)
                for (int i = 1; i < procs * 2; i++)
                {
                        if (M_max_array < MainVec[i].M)
                        {
                                M_max_array = MainVec[i].M;
                        }
                }

                // поиск наибольшего M из кучки (из M_max_cogorta)
                if (size_group != 0)
                {
                        Pointer = Local_group.begin();
                        double tmp_M;
                        double M_max_cogorta = (*Pointer++).M;

                        while (Pointer != Local_group.end())
                        {
                                tmp_M = (*Pointer).M;
                                if (M_max_cogorta < tmp_M) M_max_cogorta = tmp_M;

                                Pointer++;
                        }

                        //поиск большего M среди найденных
                        if (M_max_array < M_max_cogorta)
                        {
                                M_big = M_max_cogorta;
                        }
                        else
                        {
                                M_big = M_max_array;
                        }

                }
                else
                {
                        M_big = M_max_array;
                }
```

```
                    //подсчёт m
                    Lipschitz(M_big, m_small);

#pragma omp parallel for num_threads(procs)
                    for (int i = 0; i < procs * 2; i++)//считаем R для текущих отрезков
                    {
                           MainVec[i].R = R(m_small, MainVec[i].z_right,
MainVec[i].z_left, MainVec[i].x_new_right, MainVec[i].x_new_left);
                    }

                    if (size_group != 0)
                    {
                           std::vector<GroupTwoDim> R_vec(Local_group.begin(),
Local_group.end());
#pragma omp parallel for num_threads(procs)
                           for (int i = 0; i < size_group; i++)//подсчёт R для отрезков
из очереди
                           {
                                  R_vec[i].R = R(m_small, R_vec[i].z_right,
R_vec[i].z_left, R_vec[i].x_new_right, R_vec[i].x_new_left);
                           }
                           int j = 0;
                           for (std::list<GroupTwoDim>::iterator i = Local_group.begin();
i != Local_group.end(); i++)
                           {
                                  i->R = R_vec[j++].R;
                           }
                           R_vec.clear();
                    }
                    for (int i = 0; i < procs * 2; i++)
                    {
                           Local_group.push_back(MainVec[i]);//кладем новые группы в
общий список
                    }
                    //сбор данных
                    p.X.push_back(Local_group.back().x_right);
                    p.Y.push_back(Local_group.back().y_right);

                    Local_group.sort();
                    localSteps++;
                    if (localSteps > _Steps) { break; }//условие остановки
                    curr_left = Local_group.back().x_new_left;
                    curr_right = Local_group.back().x_new_right;
                    if (abs(curr_left - curr_right) < eps) { break; }//условие остановки
             }
      }

      end = omp_get_wtime();
      //сбор данных
      p.time = end - start;
      omp_set_dynamic(1);



      //сбор данных
      p.X.push_back(Local_group.back().x_right);
      p.Y.push_back(Local_group.back().y_right);
      p.x = Local_group.back().x_right;
      p.y = Local_group.back().y_right;
      p.z = Local_group.back().z_right;

      // очистка
      StartVec.clear();
      MainVec.clear();
```

```
	Local_group.clear();
	evolve.clear();
	delete[]x_new;
	delete[]coordinates_y;


	return p;
}

bool operator<(const GroupTwoDim& one, const GroupTwoDim& two)
{
	if (one.R < two.R)
	{
		return true;
	}
	else
	{
		return false;
	}
};

bool operator>(const GroupTwoDim& one, const GroupTwoDim& two)
{
	if (one.R > two.R)
	{
		return true;
	}
	else
	{
		return false;
	}
};
```

Simple_par.cpp

```
#include "Search.h"

void Shift_iters(borders &bord_Z, borders &bord_X, std::list<double>::iterator &iter_x,
std::list<double>::iterator &iter_z)
{
	bord_Z.left = *iter_z++;
	bord_Z.right = *iter_z;
	bord_X.left = *iter_x++;
	bord_X.right = *iter_x;
}
PointerOneDim Search::Simple_Par_searchMin(const double _left, const double _right, const
int _N_max, const double _Eps, const int _threads)
{
	procs = _threads;

	std::vector<PointerOneDim> res_threads(procs); // вектор для хранения результатов
каждого потока
	omp_set_dynamic(0);

	double start = omp_get_wtime();
	omp_set_num_threads(procs);


	if (_threads > 1)
	{

#pragma omp parallel for
		for (int i = 0; i < procs; i++)//выполняем поиск на отдельных участках
отрезка
```

```
		{
			res_threads[i] = Serial_searchMin(_left + (_right - _left)*i / procs, 
_left + (_right - _left)*(i + 1) / procs, _N_max, _Eps);

		}

		PointerOneDim res;
		res.z = res_threads[0].z;
		res.x = res_threads[0].x;
		res.steps = 0;
		res.time = 0;
		for (int i = 0; i < procs; i++)
		{
			if (res_threads[i].z < res.z)//находим меньший среди найденных
			{
				res.z = res_threads[i].z;
				res.x = res_threads[i].x;

			}
			res.time += res_threads[i].time;//суммируем общее время работы 
каждого потока
			res.steps += res_threads[i].steps;//суммируем общее число итераций 
каждого потока

			for (int j = 0; j < res_threads[i].X.size(); j++)//сохраняем 
полученные координаты в один общий вектор
			{
				res.X.push_back(res_threads[i].X[j]);
			}

		}
		return res;
	}

	else//один поток
	{

		res_threads[0] = Serial_searchMin(_left, _right, _N_max, _Eps);
		PointerOneDim res;
		res.z = res_threads[0].z;
		res.x = res_threads[0].x;
		res.time = 0;
		res.steps = res_threads[0].steps;
		for (int j = 0; j < res_threads[0].X.size(); j++)
		{

			res.X.push_back(res_threads[0].X[j]);
		}
		res.time += res_threads[0].time;
		return res;
	}


}



PointerOneDim Search::Serial_searchMin(const double _left, const double _right, const int 
_N_max, const double _Eps)
{
	PointerOneDim p;
	p.steps = 1;
	p.time = 0;
```

```
	double curr_left = _left;

	double curr_right = _right;

	double start, end;


	double z_begin = Func(curr_left); // левая граница
	double z_end = Func(curr_right); // правая граница

	double temp = 0.; // переменная для подсчёта
	int localSteps = 1; // количество шагов в Настоящем алгоритме
	int all_steps;
	double M_big;
	double m_small = 1;
	int R_max_index = 0; // Переменная индекса следующей точки испытания

	double new_X; // переменные для рабочего отрезка и точки нового испытания

	std::list<double> M_vector;
	std::list<double> R_vector;

	std::list<double> z; //значения Z
	std::list<double> x; //значения рабочих X

	borders bord_Z; // границы для Z
	borders bord_X; // границы для X

	//деление на интервалы работает так:
	//1) вставил новую точку x в текущий отрезок
	//2) новым интервалом считается получившаяся левая половина
	//3) правая половина считается старой

	std::list<double>::iterator place_M, place_R; // указатели для места установки
нового значения
	std::list<double>::iterator iter_x, iter_z, iter_R, iter_M;

	M_big = M(z_end, z_begin, curr_right, curr_left);
	Lipschitz(M_big, m_small);
	new_X = New_x(curr_right, curr_left, z_end, z_begin, m_small);


	x.push_back(curr_left);
	x.push_back(new_X);
	x.push_back(curr_right);

	iter_x = x.begin();
	for (int i = 0; i < 3; i++)
	{
		z.push_back(Func(*iter_x++));
	}

	iter_z = z.begin();
	iter_x = x.begin();




	start = omp_get_wtime();
	p.X.push_back(curr_right);
	//первая итерация
	for (int i = 1; i < 3; i++)
	{
```

```
                Shift_iters(bord_Z, bord_X, iter_x, iter_z);

                M_vector.push_back(M(bord_Z.right, bord_Z.left, bord_X.right,
bord_X.left)); // вставка новой M_big
        }

        double max = M_vector.front();

        if (max < M_vector.back()) max = M_vector.back();
        M_big = max; // поиск максимальной M_big

        Lipschitz(M_big, m_small);

        R_max_index = 1;

        iter_z = z.begin();
        iter_x = x.begin();
        double tmp;
        for (int i = 1; i < 3; i++)
        {
                Shift_iters(bord_Z, bord_X, iter_x, iter_z);

                tmp = R(m_small, bord_Z.right, bord_Z.left, bord_X.right, bord_X.left);
                R_vector.push_back(tmp);
                if (i == 1) max = tmp;

                if (max < tmp)
                {
                        max = tmp;
                        R_max_index = i;
                }
        }

        iter_z = z.begin();
        iter_x = x.begin();

        for (int i = 0; i < R_max_index; i++)
        {
                iter_z++;
                iter_x++;
        }

        curr_right = *iter_x;
        curr_left = *--iter_x;
        p.X.push_back(curr_right);

        //основной цикл
        while (true)
        {
                all_steps = 2 + localSteps;
                iter_z = z.begin(); // найдём интервалы с максимальной характеристикой
                iter_x = x.begin();
                // ищем интервалы сдвигая итераторы
                for (register int i = 0; i < R_max_index; i++)
                {
                        iter_z++;
                        iter_x++;
                }
                // забираем значения для хранения и работы нового цикла
                bord_Z.right = *iter_z--;
                bord_Z.left = *iter_z++;

                // вычисление новой точки испытания x_new
                new_X = New_x(curr_right, curr_left, bord_Z.right, bord_Z.left, m_small);
```

```
		// сохраняем x_new и z_new
		z.insert(iter_z, Func(new_X));
		x.insert(iter_x, new_X);

		iter_z = z.begin();
		iter_x = x.begin();
		place_M = M_vector.begin();
		place_R = R_vector.begin();

		// ищем интервал с лучшей R, двигая iter
		for (int i = 0; i < R_max_index - 1; i++)
		{
			iter_z++;
			iter_x++;
			if (i == R_max_index - 1)
			{
				break;
			}
			place_M++;
			place_R++;
		}

		for (int i = 0; i < 2; i++)
		{
			Shift_iters(bord_Z, bord_X, iter_x, iter_z);
			if (i == 0)
			{
				M_vector.insert(place_M, M(bord_Z.right, bord_Z.left,
bord_X.right, bord_X.left));//В новый интервал
			}
			else
			{
				*place_M = M(bord_Z.right, bord_Z.left, bord_X.right,
bord_X.left); // старый интервал
			}
		}

		iter_M = M_vector.begin();
		double temp;
		double max = *iter_M;
		temp = M_vector.back();

		for (int i = 0; i < all_steps - 1; i++)
		{
			if (max < temp) max = temp;
			temp = *++iter_M;
		}
		M_big = max;

		Lipschitz(M_big, m_small);



		iter_z = z.begin();
		iter_x = x.begin();

		R_max_index = 1;
		place_R = R_vector.begin();

		for (int i = 0; i < all_steps; i++)
		{
			Shift_iters(bord_Z, bord_X, iter_x, iter_z);
			if (i == 0)
```

```
                        {
                                R_vector.insert(place_R, R(m_small, bord_Z.right, bord_Z.left,
bord_X.right, bord_X.left));//В новый интервал
                        }
                        else
                        {
                                *place_R++ = R(m_small, bord_Z.right, bord_Z.left,
bord_X.right, bord_X.left); // старый интервал
                        }
                }

                iter_R = R_vector.begin();
                max = *iter_R;
                for (int i = 1; i < all_steps; i++)
                {
                        temp = *++iter_R;
                        if (max < temp)
                        {
                                max = temp;
                                R_max_index = i + 1;
                        }
                }

                iter_x = x.begin();
                iter_z = z.begin();

                for (register int i = 0; i < R_max_index; i++)//подвинусь до нужного
отрезка
                {
                        iter_z++;
                        iter_x++;
                }

                // сохранение
                curr_right = *iter_x;
                curr_left = *--iter_x;

                //сбор данных
                p.X.push_back(curr_right);
                p.steps++;

                if (abs(curr_left - curr_right) < _Eps) { break; } //условие выхода

                localSteps++;
                if (localSteps + 1 > _N_max) { break; }//условие выхода
        }
        end = omp_get_wtime();
        //сбор данных

        p.time = end - start;
        p.x = curr_right;
        p.z = *iter_z;
        return p;
}
```

Ch_par.cpp

```
#include "Search.h"
```

```
PointerOneDim Search::Ch_SearchMin(const double _Epsilon, const int _Steps, const int
threads)
{
	PointerOneDim p;
	p.steps = 0;
	p.time = 0;
	int procs = threads;
	int localSteps = 0; // шаги для внутреннего счета
	double eps = _Epsilon;

	//текущие границы  x[0;1]
	double curr_left;
	double curr_right;

	// время работы
	double start, end;

	double *x_new = new double[procs];
	//константы для оценки конст Липшица
	double m_small = 1;
	double M_big;

	//границы
	curr_left = left;
	curr_right = right;

	std::vector<GroupOneDim> StartVec(procs);//вектор для первой итерации

	std::vector<GroupOneDim> MainVec(procs * 2);//вектор для остальных итераций

	std::list<GroupOneDim> Local_group;

	size_t size_group; // размер list<Group> Local_group

	std::list<GroupOneDim>::iterator Pointer;
	omp_set_dynamic(0);//среда выполнения не будет динамически настроить количество
потоков
	start = omp_get_wtime();
	omp_set_num_threads(procs);//установим число нужных потоков
	{
#pragma omp parallel for num_threads(procs)
		for (int i = 0; i < procs; i++)//распределение отрезка по потокам
		{
			StartVec[i].x_left = left + (i * (right - left) / procs);
			StartVec[i].x_right = StartVec[i].x_left + (right - left) / procs;
			StartVec[i].z_left = Func(StartVec[i].x_left);
			StartVec[i].z_right = Func(StartVec[i].x_right);
		}
		// подсчет M
#pragma omp parallel for num_threads(procs)
		for (int i = 0; i < procs; i++)
		{
			StartVec[i].M = M(StartVec[i].z_right, StartVec[i].z_left,
StartVec[i].x_right, StartVec[i].x_left);
		}

		M_big = StartVec[0].M;//найденная M одним потоком

		for (int i = 1; i < procs; i++) // найдем макс из остальных
		{
			if (M_big < StartVec[i].M)
			{
				M_big = StartVec[i].M;
			}
```

```
		}
		Lipschitz(M_big, m_small);// оценка параметра m

		//подсчёт R
#pragma omp parallel for num_threads(procs)
		for (int i = 0; i < procs; i++)
		{
			StartVec[i].R = R(m_small, StartVec[i].z_right, StartVec[i].z_left,
StartVec[i].x_right, StartVec[i].x_left);
		}
		for (register int i = 0; i < procs; i++)//вставка результатов в общий
список
		{
			Local_group.push_back(StartVec[i]);
		}
		Local_group.sort();//сортировка по возрастанию
		//получение текущих границ
		curr_left = Local_group.back().x_left;
		curr_right = Local_group.back().x_right;
		localSteps++;
		while (true)//основной цикл
		{
			Pointer = --Local_group.end();
			for (int i = 0; i < procs * 2 - 2; i += 2) // хватаем proc_count R-ок
(самых больших)
			{
				MainVec[i] = *Pointer;
				Local_group.erase(Pointer--);//остальные R-ки стираем (все
группы)
			}
			MainVec[procs * 2 - 2] = *Pointer;
			Local_group.erase(Pointer);//.erase сотрет по указателю
			size_group = Local_group.size();

			// соответствующей  кучке больших R-ок находим соответствующие точки
испытаний
#pragma omp parallel for shared(x_new) num_threads(procs)
			for (int i = 0; i < procs; i++)
			{
				x_new[i] = New_x(MainVec[i * 2].x_right, MainVec[i *
2].x_left, MainVec[i * 2].z_right, MainVec[i * 2].z_left, m_small);
				MainVec[i * 2 + 1].z_left = Func(x_new[i]);
			}
			for (register int i = 0; i < procs * 2; i += 2)//сохраняем найденные
точки
			{
				MainVec[i + 1].x_right = MainVec[i].x_right;
				MainVec[i + 1].z_right = MainVec[i].z_right;
				MainVec[i + 1].x_left = MainVec[i].x_right = x_new[i / 2];
				MainVec[i].z_right = MainVec[i + 1].z_left;
			}
			// считаем M
#pragma omp parallel for num_threads(procs)
			for (int i = 0; i < procs * 2; i++)
			{
				MainVec[i].M = M(MainVec[i].z_right, MainVec[i].z_left,
MainVec[i].x_right, MainVec[i].x_left);
			}

			double M_max_all_M = MainVec[0].M;//найденная M одним потоком


				//поиск наибольшего M, который не соответсвует текущей
кучке(из M_max_all_M) (а он может быть большим, чем все M из кучки)
```

```
                    for (int i = 1; i < procs * 2; i++)
                    {
                           if (M_max_all_M < MainVec[i].M)
                           {
                                  M_max_all_M = MainVec[i].M;
                           }
                    }
                    // поиск наибольшего M из кучки (из M_max_cogorta)
                    if (size_group != 0)
                    {
                           Pointer = Local_group.begin();
                           double tmp_M;
                           double M_max_cogorta = (*Pointer++).M;
                           while (Pointer != Local_group.end())
                           {
                                  tmp_M = (*Pointer).M;
                                  if (M_max_cogorta < tmp_M) M_max_cogorta = tmp_M;
                           Pointer++;
                           }
                           //поиск большего M среди найденных
                           if (M_max_all_M < M_max_cogorta)
                           {
                                  M_big = M_max_cogorta;
                           }
                           else
                           {
                                  M_big = M_max_all_M;;
                           }
                    }
                    else
                    {
                           M_big = M_max_all_M;
                    }
                    //подсчёт m
                    Lipschitz(M_big, m_small);

#pragma omp parallel for num_threads(procs)
                    for (int i = 0; i < procs * 2; i++)//считаем R для текущих отрезков
                    {
                           MainVec[i].R = R(m_small, MainVec[i].z_right,
MainVec[i].z_left, MainVec[i].x_right, MainVec[i].x_left);
                    }


                    if (size_group != 0)
                    {
                           std::vector<GroupOneDim> R_vec(Local_group.begin(),
Local_group.end());

#pragma omp parallel for num_threads(procs)
                           for ( int i = 0; i < size_group; i++)//подсчёт R для отрезков
из очереди
                           {
                                  R_vec[i].R = R(m_small, R_vec[i].z_right,
R_vec[i].z_left, R_vec[i].x_right, R_vec[i].x_left);
                           }
                           int j = 0;
                           for (std::list<GroupOneDim>::iterator i = Local_group.begin();
i != Local_group.end(); i++)
                           {
                                  i->R = R_vec[j++].R;
                           }
                           R_vec.clear();
                    }
```

```
                    for ( int i = 0; i < procs * 2; i++)
                    {

                           Local_group.push_back(MainVec[i]);//кладем новые группы в
общий список
                    }

                    //сбор данных
                    p.X.push_back(Local_group.back().x_right);//.back() возвращает ссылку
на последний элемент в контейнере списка
                    p.steps++;

                    Local_group.sort();

                    localSteps++;
                    if (localSteps > _Steps) { break; }//условие остановки

                    curr_left = Local_group.back().x_left;
                    curr_right = Local_group.back().x_right;
                    if (abs(curr_left - curr_right) < eps) { break; }//условие остановки
             }
      }
      end = omp_get_wtime();
      //сбор данных
      p.time = end - start;

      omp_set_dynamic(1);
      //сбор данных
      p.X.push_back(Local_group.back().x_right);
      p.x = curr_right;
      p.z = Local_group.back().z_right;
      // очистка
      StartVec.clear();
      MainVec.clear();
      delete[] x_new;

      return p;
}

bool operator<(const GroupOneDim& one, const GroupOneDim& two)
{
      if (one.R < two.R)
      {
            return true;
      }
      else
      {
            return false;
      }
};

bool operator>(const GroupOneDim& one, const GroupOneDim& two)
{
      if (one.R > two.R)
      {
            return true;
      }
      else
      {
            return false;
      }
};
```