МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

**«Нижегородский государственный университет им. Н.И. Лобачевского»**

**Национальный исследовательский университет**

**Институт информационных технологий математики и механики**

**Отчет по лабораторной работе**

**«Алгоритм глобального поиска (Стронгина) для одномерных задач оптимизации. Распараллеливание по характеристикам»**

**Выполнил**: студент группы 381508

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Исрафилов М. Ш.

Подпись

**Проверил:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Сысоев А. В.

Подпись

Нижний Новгород

2018

Оглавление

[Введение 3](#_Toc514699758)

[Постановка задачи 4](#_Toc514699759)

[Тестовая версия 5](#_Toc514699760)

[Последовательная версия алгоритма 5](#_Toc514699761)

[Генератор 10](#_Toc514699762)

[Чекер 10](#_Toc514699763)

[Параллельные версии 12](#_Toc514699764)

[OpenMP 12](#_Toc514699765)

[TBB 13](#_Toc514699766)

[Подтверждение корректности 17](#_Toc514699767)

[Результаты экспериментов 18](#_Toc514699768)

[Заключение 19](#_Toc514699769)

**Введение**

Алгоритм глобального поиска (Стронгина) – характеристический метод решения задач глобальной оптимизации. Поиск производиться до тех пор, пока либо разница между двумя последними значениями не достигнет заданного числа, либо количество итераций не достигнет максимума.

Задача глобальной оптимизации – одномерная, многоэкстремальная, липшицева функция с рядом дополнительных характеристик, таких как:

* Область определения
* Число итераций поиска
* Константа точности
* Константа надежности алгоритма

**Постановка задачи**

Дана задача оптимизации. Алгоритм проводит вычисление функции на каждой итерации. На первых двух итерациях алгоритм вычисляет функцию на концах отрезка, далее на каждой итерации:

1. На основе предыдущих итераций вычисляется максимальная оценка константы Липшица и умножается на константу надежности алгоритма
2. Вычисляется интервал с максимальной характеристикой.
3. Ставиться точка внутри интервала
4. Вычисляется функция в новой точке
5. Точки сортируются
6. Между двумя последними вычисленными точками проверяется разница
7. Если разница больше заданной в задаче или превышено число итераций, то дальнейшие вычисления приостанавливаются и в качестве оптимальной берется последняя вычисленная точка, иначе продолжается вычисление.

**Тестовая версия**

Тестовая версия принимает на вход бинарный файл задачи и выполняет последовательную версию алгоритма.

Результат записывается в выходной файл, название которого формируется из названия входного файла с добавлением «.usans»

Последовательная версия алгоритма

Последовательная версия реализует алгоритм представленный в постановке задачи.

Алгоритм начинается с функции:

Point CalculateOptimum(Task\*);

На вход поступает структура задачи, на выходе получается структура точки экстремума.

struct Task

{

int funcNum;

int maxOfIterations;

double xl;

double xr;

double eps;

int r;

};

struct Point

{

double x;

double y;

bool err;

};

bool err – значение, которое обозначает ошибку вычисления.

В качестве хранения промежуточных результатов испытания, взят контейнер map<double, double> для хранения точек в отсортированном виде, т.к. этого требует алгоритм.

Функции, используемые в задаче, берутся из:

double Function(double x, int funcNum)

{

double result;

switch (funcNum)

{

case 0:

result = cos(x)\*(x/4);

break;

case 1:

result = sin(x)\*(x/4);

break;

case 2:

result = sqrt(pow(x,2));

break;

case 3:

result = exp(x);

break;

case 4:

result = x;

break;

case 5:

result = pow(x,3);

break;

case 6:

result = 7.0\*pow(x,2)+3.5\*x-1.4;

break;

case 7:

result = sqrt(x)-12.4;

break;

case 8:

result = atan(x);

break;

case 9:

result = 2.1\*x/(pow(x,2)+1.2);

break;

case 10:

result = 1/(1+pow(x,2));

break;

case 11:

result = 31\*pow(x, 4)+7\*pow(x,3)-2\*pow(x,2)+x;

break;

case 12:

result = -sqrt((1-x)/(1+x));

break;

case 13:

result = sqrt((1-x)/(1+x));

break;

case 14:

result = (3.8\*pow(x,2)+6.2\*x+2.7)/(2.3\*x+1.9);

break;

case 15:

result = 7;

break;

case 16:

result = sqrt((pow(x,2)-1)\*(9-pow(x,2)));

break;

case 17:

result = x+1/x;

break;

case 18:

result = (1/(1+x))-(2/pow(x, 2))+(1/(1-x));

break;

case 19:

result = (x+1)\*(x-2)/(x-1)\*(x+2);

break;

default:

result = NAN;

}

return result;

}

В качестве промежуточного результата берется наименьшая точка из двух испытаний. Далее в цикле происходит дальнейшие вычисления.

typedef map<double, double> mapP;

Characteristic - вычисляет левую границу интервала с наибольшей характеристикой.

mapP::iterator Characteristic(mapP& p, double m)

{

mapP::iterator interval;

double res, tmp;

for (auto itPL = p.begin(), itPR = ++p.begin(); itPR != p.end(); ++itPL, ++itPR)

{

if (itPL == p.begin())

{

res = m \* (itPR->first - itPL->first) + (pow(itPR->second - itPL->second, 2) /

(m \* (itPR->first - itPL->first))) - 2 \* (itPR->second + itPL->second);

interval = itPL;

}

else

{

tmp = m \* (itPR->first - itPL->first) + (pow(itPR->second - itPL->second, 2) /

(m \* (itPR->first - itPL->first))) - 2 \* (itPR->second + itPL->second);

if (res < tmp)

{

res = tmp;

interval = itPL;

}

}

}

return interval;

}

double GetM(mapP&, int) - вычисляет наибольшую оценку константы Липшица умноженную на константу надёжности r.

double SetPoint(mapP::iterator, mapP::iterator, double) – устанавливает точку в интервале с наибольшей характеристикой.

bool Distance(mapP::iterator, double) – проверяет разницу между двумя последними вычисленными точками с константой eps.

**Генератор**

На вход получает целое число - количество тестов, которые нужно сгенерировать. На выходе получаются 2 файла - задача и эталонный ответ.

Имя файла задачи равно номеру теста: 1, 2, 3 и т.д. Имя файла ответа: 1.ans, 2.ans, 3.ans и т.д.

Генерируются:

* Номер функции от 0 до 20 (21 различных вариантов), функций всего 20, 21 вариант нужен для проверки алгоритма на корректную работу при ошибке вычисления.
* Количество итераций от 100 до 10000
* Границы от -100 до 100
* Оценка точности от 0.1 до 1e-10
* Константа надежности от 1 до 10

**Чекер**

Программа проверки. На вход получает имя файла. Имя файла ответа пользователя и имя файла эталонного ответа должны совпадать без расширения. Результат выводится в файл result.txt.

Например: 1.ans(ответ-эталон), 1.usans(пользовательский ответ).

**Параллельные версии**

Параллельные версии отличаются от последовательной несколькими аспектами:

* Параллельные версии могут работать только с контейнерами, у которых доступны итераторы с произвольным доступом. В качестве него взят vector<pair<double, double>>.
* Изменена функция вычисления характеристик.
* На вход поступает не только имя файла задачи, но и число потоков

Цикл вычисляющий наибольшую характеристику разделяется между потоками, после чего возвращается в качестве результата левая граница интервала с этой характеристикой.

**OpenMP**

Параллельное вычисление характеристик:

int Characteristic(vecP& p, double m)

{

int leftBorder;

double res, tmp;

res = m \* (p[1].first - p[0].first) + (pow(p[1].second - p[0].second, 2) /

(m \* (p[1].first - p[0].first))) - 2 \* (p[1].second + p[0].second);

leftBorder = 0;

#pragma omp parallel for shared(leftBorder, res, p) private (tmp)

for (int i = 1; i < p.size()-1; ++i)

{

tmp = m \* (p[i+1].first - p[i].first) + (pow(p[i+1].second - p[i].second, 2) /

(m \* (p[i+1].first - p[i].first))) - 2 \* (p[i+1].second + p[i].second);

if (res < tmp)

{

res = tmp;

leftBorder = i;

}

}

return leftBorder;

}

**TBB**

Вместо цикла в функции вычисления характеристики используется функция tbb::parallel\_reduce, а цикл из функции Characteristic переходит в класс tbbCharacteristics

int Characteristic(vecP& p, double m)

{

int leftBorder;

double res, tmp;

res = m \* (p[1].first - p[0].first) + (pow(p[1].second - p[0].second, 2) /

(m \* (p[1].first - p[0].first))) - 2 \* (p[1].second + p[0].second);

leftBorder = 0;

tbbCharacteristics tbbRes(p, res, leftBorder, m);

parallel\_reduce(blocked\_range<int>(1, p.size() - 1), tbbRes);

if (res < tbbRes.getRes()) {

res = tbbRes.getRes();

leftBorder = tbbRes.getBorder();

}

return leftBorder;

}

Класс tbbCharacteristics:

typedef vector<pair<double, double> > vecP;

class tbbCharacteristics {

const vecP p;

double res, tmp, leftBorder, m;

public:

explicit tbbCharacteristics(vecP& \_vec, double \_res, double \_leftBorder, double \_m) :

p(\_vec), res(\_res), leftBorder(\_leftBorder), m(\_m) {}

tbbCharacteristics(const tbbCharacteristics& c, split) : p(c.p), res(c.res), leftBorder(c.leftBorder) {}

void operator() (const blocked\_range<int>& r) {

int begin = r.begin(), end = r.end();

for (int i = begin; i != end; i++) {

tmp = m \* (p[i + 1].first - p[i].first) + (pow(p[i + 1].second - p[i].second, 2) /

(m \* (p[i + 1].first - p[i].first))) - 2 \* (p[i + 1].second + p[i].second);

if (res < tmp)

{

res = tmp;

leftBorder = i;

}

}

}

void join(const tbbCharacteristics& tbbChar) {

if (res < tbbChar.res) {

res = tbbChar.res;

leftBorder = tbbChar.leftBorder;

}

}

double getRes() {

return res;

}

double getBorder() {

return leftBorder;

}

};

**Подтверждение корректности**

Корректность алгоритма подтверждена на 10 тестах, представленных в тестовой версии.

**Результаты экспериментов**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 1 | 2 | 4 | 8 |
| Serial | 8,882795 |  |  |  |
| OpenMP |  | 3,407538 | 3,092561 | 3,923559 |
| TBB |  | 3,26685 | 3,211334 | 3,453531 |

Проверка проводилась на функции sin(x)\*(x/4) с большой константой надежности и достаточно малой константой погрешности, с областью определения [-50; 150].

**Заключение**

В данной работе были рассмотрены библиотеки и стандарты параллельного программирования такие, как OpenMP и TBB на примере реализации алгоритма глобального поиска (Стронгина) для одномерных задач оптимизации.