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

Отчет по лабораторной работе

«Многошаговая схема решения двумерных задач глобальной оптимизации. Распараллеливание по характеристикам»

Выполнил:
студент группы 381706-2
Назаров В.В.

Проверил:
доцент кафедры МОСТ,
Сысоев А.В.

Нижний Новгород
2019

Содержание

Введение

Глобальная оптимизация - это раздел прикладной математики и численного анализа, который занимается проблемами поиска глобальных экстремумов функций. В большинстве случаев решается задача минимизации, поскольку максимизация эквивалентна поиску обратного функционала для задачи минимизации. Приведем примеры известных алгоритмов глобальной оптимизации:

- Метод последовательного сканирования
- Метод Кушнера
- Метод Ломаных
- Базовый информационно-статистический алгоритм глобального поиска (АГП)
- Информационный алгоритм глобального поиска с испытаниями только во внутренних точках (АГПВТ)

Для снижения сложности алгоритмов глобальной оптимизации, формирующих неравномерное покрытие области поиска, широко используются различные схемы редукции размерности, которые позволяют свести решение многомерных оптимизационных задач к семейству задач одномерной оптимизации.

В данной работе рассматривается рекурсивная схема редукции размерности с использованием базового информационно-статистического алгоритма глобального поиска.

Целью лабораторной работы является реализация параллельной многошаговой схемы решения двумерных задач глобальной оптимизации с распараллеливанием по характеристикам и сравнение эффективности параллельной и последовательной схемы.

Постановка задачи

Необходимо реализовать последовательную и параллельную рекурсивную схему редукции размерности с использованием базового информационно-статистического алгоритма глобального поиска, провести тестирование работоспособности реализованных алгоритмов, а также сравнить время выполнения последовательной и параллельной схемы. Распараллеливание выполнить по максимальным характеристикам. Сделать выводы по полученным результатам.

Описание алгоритма

Для решения двумерной задачи глобальной оптимизации используется базовый алгоритм глобального поиска (АГП) на двух уровнях рекурсии. На первом уровне с помощью АГП определяется в какой точке необходимо провести испытание, точка фиксируется и отправляется на второй уровень, где определяется результат испытания и возвращается полученное значение.

Рассмотрим общую схему АГП:

Пусть проведены k >= 2 испытаний и получена поисковая информация $\omega = \omega_k = \{(x_i, z_i),\ 1 \leqslant i \leqslant k\}$ для выбора точки x^{k+1} нового испытания необходимо выполнить следующие действия:

1. Упорядочить точки x^i в порядке возрастания
2. Вычислить величины M и m, где $M = \max\limits_{1 \leqslant i \leqslant k-1} \left| \frac{z_i - z_{i-1}}{x_i - x_{i-1}} \right|$, $m = \begin{cases} rM, & M > 0, \\ 1, & M = 0, \end{cases}$ где r > 1 является заданным параметром метода
3. Для каждого интервала (x_{i-1}, x_i), 1 <= i <= 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 с максимальной характеристикой
5. Провести новое испытание в точке x^{k+1}, вычислить значение z^{k+1} и увеличить номер шага поиска k на единицу, $x^{k+1} = \frac{1}{2}(x_t + x_{t-1}) - \frac{z_t - z_{t-1}}{2m}$

Правило остановки задается в форме

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

где ε > 0 – заданная точность поиска.

Схема распараллеливания

В данной лабораторной работе используется распараллеливание по характеристикам. Пусть количество процессов равно n, процесс с рангом 0 назовем главным, а остальные вспомогательными. В начале работы процесс с рангом 0 делит отрезок [a, b] на необходимое количество интервалов k-1, в зависимости от количества процессов, и параллельно вычисляются значения z^i во всех точках i, $1 < i < k$, полученного множества. После того, как получены и отсортированы все необходимые элементы множества, начинается k+1 итерация: Процесс с рангом 0 вычисляет характеристики R, все значения характеристик сортируются в порядке убывания и на основании первых n-1 значений R вычисляются точки, в которых вспомогательные процессы должны провести испытания. Если после получения главным процессом результатов всех испытаний, правило остановки != 0, начинается следующая итерация, иначе вспомогательным процессам рассылается сигнал о завершении работы и возвращается финальный результат главным процессом.

Описание программной реализации

В программе реализованы три класса для поддержки элементов множества с перегруженной операцией сравнения, структура и вспомогательная функция для поддержки возвращаемого результата:

- class setElemOneVar – класс, поддерживающий элемент множества в виде одномерной функции
- class setElemTwoVar – класс, поддерживающий элемент множества в виде двумерной функции
- class setElemR – класс, поддерживающий элемент множества в виде характеристики R
- struct resultTwoVar – структура, поддерживающая результат в виде двумерной функции
- bool compareResults(const resultTwoVar& a, const resultTwoVar& b, const double& eps = 0.01) – функция сравнения двух результатов с заданной точностью eps

Также реализованы три основные функции для решения двумерных задач глобальной оптимизации, каждая из которых принимает саму функцию func, область поиска, точность поиска eps, параметр метода r_par и максимальное количество итераций N_max:

- resultTwoVar solveOneVar(const double& _a, const double& _b, const double& Xf, double(*func)(double x, double y), const double& _eps = 0.1, const int& _N_max = 100, const double& _r_par = 2.0) – функция, которая находит глобальный минимум одномерной функции на отрезке [_a, _b] с фиксированным Xf
- resultTwoVar solveTwoVar(const double& _a1, const double& _b1, const double& _a2, const double& _b2, double(*func)(double x, double y), const double& _eps = 0.1, const int& _Nmax = 100, const double& _epsOneVar = 0.1, const int& _NmaxOneVar = 100, const double& _r_par = 2.0) – параллельное решение двумерной задачи глобальной оптимизации
  - resultTwoVar solveTwoVarSequential(const double& _a1, const double& _b1, const double& _a2, const double& _b2, double(*func)(double x, double y), const double& _eps = 0.1, const int& _Nmax = 100, const double& _epsOneVar = 0.1, const int& _NmaxOneVar = 100, const double& _r_par = 2.0) - последовательное решение двумерной задачи глобальной оптимизации

Подтверждение корректности

Для подтверждения корректности в программе представлен набор из шести тестов, разработанных с помощью использования Google C++ Testing Framework.

Корректность проверяется на шести различных функциях. Результаты параллельного алгоритма сравниваются либо с последовательным алгоритмом, либо с заранее заданными значениями, которые были отдельно посчитаны. Приведу пример нескольких функций:

func = $4 + \sqrt[3]{(x^2 + y^2)^2}$, x ∈ (-0.9, 1), y ∈ (-1, 1), результат = (0, 0, 4)

func = $x + 4y - 6 - 2\log(xy) - 3\log(y)$, x ∈ (1, 3), y ∈ (1, 2), результат = (2, 1.25, -1.5)

func = $2\sin(x) + \cos(y)$, x ∈ (1, 5), y ∈ (0.5, 4), результат = (3*π/2, π, -3)

Успешное прохождение тестов доказывает корректность выполнения алгоритма.

Результаты экспериментов

Эксперименты проводились на оборудовании со следующей аппаратной конфигурацией:

- Процессор: Intel Core i7-3537U CPU @ 2.00 GHz
- Оперативная память: 8 GB
- Версия MPI: Open MPI 3.1.3
- Операционная система: Ubuntu 19.10

В таблице 1 приведена зависимость времени работы алгоритма при различных интервалах и различном количестве процессов. Точность поиска $\varepsilon = 0.01$, максимальное количество итераций = 2500, параметр метода r = 2, функция = $x^2 + (y-1)^2$

Интервал	Последовательный алгоритм	Параллельный алгоритм					
		2 процесса		3 процесса		4 процесса	
		время	ускорение	время	ускорение	время	ускорение
x ∈ (-1.1, 1) y ∈ (-1, 1.1)	1.403	1.399	1.003	1.163	1.206	0.763	1.839
x ∈ (-3.2, 3.1) y ∈ (-3, 3.1)	22.409	22.261	1.007	17.074	1.312	11.934	1.878
x ∈ (-5.2, 5.3) y ∈ (-5.1, 5.2)	83.699	83.689	1.0001	64.651	1.295	46.547	1.798
x ∈ (-7.1, 7) y ∈ (-7.2, 7.1)	222.129	222.09	1.0001	164.05	1.354	121.45	1.829

Таблица 1. Результаты вычислительных экспериментов.

По результатам экспериментов видно, что параллельный алгоритм работает быстрее последовательного. Ускорение возрастает при увеличении числа процессов. При увеличении размера интервала, время работы возрастает, но ускорение при различном количестве итераций почти не меняется. На двух процессах ускорения почти нет, потому что поиском глобального минимума занимается один вспомогательный процесс, а главный процесс ищет минимум единственный раз при инициализации множества.

Заключение

В результате лабораторной работы была реализована многошаговая схема решения двумерных задач глобальной оптимизации, распараллеленная по характеристикам с помощью средств MPI. Результаты экспериментов показали, что параллельная версия работает быстрее последовательного. Для подтверждения корректности работы программы был разработан набор тестов с использованием библиотеки модульного тестирования Google C++ Testing Framework.

Литература

- Стронгин Р. Г., Гергель В. П., Гришагин В. А., Баркалов К. А. Параллельные вычисления в задачах глобальной оптимизации: Монография / Предисл.: В. А. Садовничий. – М.: Издательство Московского университета, 2013. – 280 с., илл. – (Серия «Суперкомпьютерное образование») ISBN 978-5-211-06479-9

Приложение

global_optimization.h

```
// Copyright 2019 Nazarov Vladislav
#ifndef MODULES_TASK_3_NAZAROV_V_GLOBAL_OPTIMIZATION_GLOBAL_
OPTIMIZATION_H_
#define MODULES_TASK_3_NAZAROV_V_GLOBAL_OPTIMIZATION_GLOBAL_
OPTIMIZATION_H_

#include <cmath>

double f1(double x, double y);
double f2(double x, double y);
double f3(double x, double y);
double f4(double x, double y);
double f5(double x, double y);
double f6(double x, double y);

class setElemOneVar {
 public:
    const double x;
    const double y;
    setElemOneVar(double _x, double _y) : x(_x), y(_y) {}
    friend bool operator<(const setElemOneVar& l, const setElemOneVar& r){
        return l.x < r.x;
    }
};

class setElemTwoVar {
 public:
    const double x;
    const double y;
    const double z;
    setElemTwoVar(const double _x, const double _y, const double _z = 0) : x(_x), y(_y), z(_z) {}
```

```
    friend bool operator<(const setElemTwoVar& l, const setElemTwoVar& r){
      return l.x < r.x;
    }
};

class setElemR {
 public:
    const double R;
    const double x;
    const double z;
    const double xPrev;
    const double zPrev;
    setElemR(const double _R, const double _x, const double _z, const double _xPrev,
      const double _zPrev) :   R(_R), x(_x), z(_z), xPrev(_xPrev), zPrev(_zPrev) {}
    friend bool operator<(const setElemR& l, const setElemR& r) {
      return l.R > r.R;
    }
};

struct resultTwoVar {
    double x;
    double y;
    double z;
};

bool compareResults(const resultTwoVar& a, const resultTwoVar& b, const double& eps = 0.01);

resultTwoVar solveOneVar(const double& _a, const double& _b,
  const double& Xf, double(*func)(double x, double y), const double& _eps = 0.1,
  const int& _N_max = 100, const double& _r_par = 2.0);

resultTwoVar solveTwoVar(const double& _a1, const double& _b1, const double& _a2,
  const   double& _b2, double(*func)(double x, double y), const double& _eps = 0.1,
  const int& _Nmax = 100, const double& _epsOneVar = 0.1, const int& _NmaxOneVar = 100,
  const double& _r_par = 2.0);
```

```
resultTwoVar solveTwoVarSequential(const double& _a1, const double& _b1, const double& _a2,
  const double& _b2, double(*func)(double x, double y), const double& _eps = 0.1, const int&
  _Nmax = 100, const double& _epsOneVar = 0.1, const int& _NmaxOneVar = 100,
  const double& _r_par = 2.0);

#endif  // MODULES_TASK_3_NAZAROV_V_GLOBAL_OPTIMIZATION_
GLOBAL_OPTIMIZATION_H_
```

global_optimization.cpp

```
// Copyright 2019 Nazarov Vladislav
#include <mpi.h>
#include <stdexcept>
#include <set>
#include <random>
#include "../../../modules/task_3/nazarov_v_global_optimization/global_optimization.h"

double f1(double x, double y) {
   return std::pow(x, 2) + std::pow(y - 1, 2);  // 0 0 0
}

double f2(double x, double y) {
   return 4 + std::pow(std::pow(std::pow(x, 2) + std::pow(y, 2), 2), 1.0/3);  // 0 0 4
}

double f3(double x, double y) {  // 1 1/2 4
   return std::pow(x, 3) + 8*std::pow(y, 3) - 6*x*y + 5;
}
double f4(double x, double y) {  // no min
   return y*sqrt(x) - 2*std::pow(y, 2) - x + 14*y;
}
double f5(double x, double y) {
   return x + 4*y - 6 - 2*log(x*y) - 3*log(y);  // 2 1.25 z = -2*ln(5/2)-3*ln(5/4)+1 = -1.5
}
```

```
double f6(double x, double y) {
    return 2*sin(x) + cos(y);  // 3*pi/2 pi -3
}

resultTwoVar solveTwoVar(const double& _a1, const double& _b1, const double& _a2,
  const double& _b2, double(*func)(double x, double y), const double& _eps, const int& _Nmax,
  const double& _epsOneVar, const int& _NmaxOneVar, const double& _r_par) {
    if (_a1 > _b1)
        throw "A1 > B1";
    if (_a2 > _b2)
        throw "A2 > B2";
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size < 2)
        return solveTwoVarSequential(_a1, _b1, _a2, _b2, func);
    MPI_Status status;
    resultTwoVar finalRes = {0, 0, 0};
    resultTwoVar res;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        std::set<setElemTwoVar> set;
        int k;
        if (std::abs(_a1 - _b1) <= 0.0001 || std::abs(_a2 - _b2) <= 0.0001) {
            k = size + 1;
        } else {
            k = size;
        }
        double segmentLen = (_b1 - _a1) / (k - 1);
        for (int i = 0; i < size - 1; ++i) {
            double Xf = _a1 + i * segmentLen;
            MPI_Send(&Xf, 1, MPI_DOUBLE, i+1, 1, MPI_COMM_WORLD);
        }
        res = solveOneVar(_a2, _b2, _b1, func, _epsOneVar);
        set.insert(setElemTwoVar(res.x, res.y, res.z));
```

```
finalRes = res;
if (k != size) {
    res = solveOneVar(_a2, _b2, _a1 + segmentLen * size, func, _epsOneVar);
    set.insert(setElemTwoVar(res.x, res.y, res.z));
    if (res.z < finalRes.z)
        finalRes = res;
}
for (int i = 0; i < size - 1; ++i) {
    MPI_Recv(&res, 3, MPI_DOUBLE, MPI_ANY_SOURCE, 1,
                        MPI_COMM_WORLD, &status);
    if (res.z < finalRes.z)
        finalRes = res;
    set.insert(setElemTwoVar(res.x, res.y, res.z));
}
std::set<setElemR> setR;
double M, currM, m, currR, newX;
bool terminate = false;
while (!terminate && k < _Nmax) {
    setR.clear();
    M = -1;
    auto iter = set.begin();
    iter++;
    auto prevIter = set.begin();
    while (iter != set.end()) {
        currM = std::abs(static_cast<double>((iter->z - prevIter->z) / (iter->x - prevIter->x)));
        if (currM > M)
            M = currM;
        iter++;
        prevIter++;
    }
    if (M > 0)
        m = _r_par * M;
    else
        m = 1;
    iter = set.begin();
```

```
            iter++;
            prevIter = set.begin();
            while (iter != set.end()) {
                currR = m*(iter->x - prevIter->x) + (std::pow((iter->z - prevIter->z), 2) /
                (m * (iter->x - prevIter->x))) - 2 * (iter->z - prevIter->z);
                setR.insert(setElemR(currR, iter->x, iter->z, prevIter->x, prevIter->z));
                iter++;
                prevIter++;
            }
            auto itR = setR.begin();
            for (int i = 0; i < size - 1; ++i) {
                k++;
                newX = (0.5)*(itR->x + itR->xPrev) - ((itR->z - itR->zPrev) / (2 * m));
                MPI_Send(&newX, 1, MPI_DOUBLE, i+1, 1, MPI_COMM_WORLD);
                if (itR->x - itR->xPrev <= _eps) {
                    terminate = true;
                }
                itR++;
            }
            for (int i = 0; i < size - 1; ++i) {
                MPI_Recv(&res, 3, MPI_DOUBLE, MPI_ANY_SOURCE, 1,
                            MPI_COMM_WORLD, &status);
                if (res.z < finalRes.z)
                    finalRes = res;
                set.insert(setElemTwoVar(res.x, res.y, res.z));
            }
        }
        for (int i = 0; i < size - 1; ++i) {
            double terminate = _eps*0.001;
            MPI_Send(&terminate, 1, MPI_DOUBLE, i+1, 1, MPI_COMM_WORLD);
        }
    } else {
        bool terminate = false;
        while (!terminate) {
            double mes;
```

```
            MPI_Recv(&mes, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, &status);
            if (mes == _eps*0.001) {
                terminate = true;
            } else {
                res = solveOneVar(_a2, _b2, mes, func, _epsOneVar, _NmaxOneVar);
                MPI_Send(&res, 3, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
            }
        }
    }
    return finalRes;
}

resultTwoVar solveTwoVarSequential(const double& _a1, const double& _b1, const double& _a2,
  const double& _b2, double(*func)(double x, double y), const double& _eps, const int& _Nmax,
  const double& _epsOneVar, const int& _NmaxOneVar, const double& _r_par) {
    if (_a1 > _b1)
        throw "A1 > B1";
    if (_a2 > _b2)
        throw "A2 > B2";
    resultTwoVar finalRes = {0, 0, 0};
    resultTwoVar res;
    std::set<setElemTwoVar> set;
    res = solveOneVar(_a2, _b2, _a1, func, _epsOneVar, _NmaxOneVar);
    set.insert(setElemTwoVar(res.x, res.y, res.z));
    finalRes = res;
    res = solveOneVar(_a2, _b2, _b1, func, _epsOneVar, _NmaxOneVar);
    set.insert(setElemTwoVar(res.x, res.y, res.z));
    if (res.z < finalRes.z)
        finalRes = res;
    double M, currM, m, currR, newX;
    int k = 2;
    std::set<setElemR> setR;
    bool terminate = false;
    while (!terminate && k < _Nmax) {
        setR.clear();
```

```
    M = -1;
    auto iter = set.begin();
    iter++;
    auto prevIter = set.begin();
    while (iter != set.end()) {
        currM = std::abs(static_cast<double>((iter->z - prevIter->z) / (iter->x - prevIter->x)));
        if (currM > M)
            M = currM;
        iter++;
        prevIter++;
    }
    if (M > 0)
        m = _r_par * M;
    else
        m = 1;
    iter = set.begin();
    iter++;
    prevIter = set.begin();
    while (iter != set.end()) {
        currR = m*(iter->x - prevIter->x) + (std::pow((iter->z - prevIter->z), 2) /
        (m * (iter->x - prevIter->x))) - 2 * (iter->z - prevIter->z);
        setR.insert(setElemR(currR, iter->x, iter->z, prevIter->x, prevIter->z));
        iter++;
        prevIter++;
    }
    k++;
    auto Riter = setR.begin();
    newX = (0.5)*(Riter->x + Riter->xPrev) - ((Riter->z - Riter->zPrev) / (2 * m));
    res = solveOneVar(_a2, _b2, newX, func, _epsOneVar, _NmaxOneVar);
    set.insert(setElemTwoVar(res.x, res.y, res.z));
    if (res.z < finalRes.z)
        finalRes = res;
    if (Riter->x - Riter->xPrev <= _eps)
        terminate = true;
}
```

```
    return finalRes;
}

resultTwoVar solveOneVar(const double& _a, const double& _b, const double& Xf,
  double(*func)(double x, double y),  const double& _eps, const int& _N_max,
  const double& _r_par) {
    if (_a > _b)
        throw "A > B";
    bool timeToStop = false;
    std::set<setElemOneVar> set;
    resultTwoVar res;
    res.x = Xf;
    double M, currM, m, R, currR, newX;
    double funcTmp = func(Xf, _a);
    set.insert(setElemOneVar(_a, funcTmp));
    res.y = _a;
    res.z = funcTmp;
    funcTmp = func(Xf, _b);
    set.insert(setElemOneVar(_b, funcTmp));
    if (res.z > funcTmp) {
        res.y = _b;
        res.z = funcTmp;
    }
    int k = 2;
    auto maxRiter = set.begin();
    auto maxPrevRiter = set.begin();
    while (!timeToStop && k < _N_max) {
        M = -1;
        auto iter = set.begin();
        iter++;
        auto prevIter = set.begin();
        while (iter != set.end()) {
            currM = std::abs(static_cast<double>((iter->y - prevIter->y) / (iter->x - prevIter->x)));
            if (currM > M)
                M = currM;
```

```
            iter++;
            prevIter++;
        }
        if (M > 0)
            m = _r_par * M;
        else
            m = 1;
        iter = set.begin();
        iter++;
        prevIter = set.begin();
        R = -200000000;
        while (iter != set.end()) {
            currR = m*(iter->x - prevIter->x) + (std::pow((iter->y - prevIter->y), 2) /
            (m * (iter->x - prevIter->x))) - 2 * (iter->y - prevIter->y);
            if (currR > R) {
                R = currR;
                maxRiter = iter;
                maxPrevRiter = prevIter;
            }
            iter++;
            prevIter++;
        }
        k++;
        newX = (0.5)*(maxRiter->x + maxPrevRiter->x) - ((maxRiter->y - maxPrevRiter->y) /
                    (2 * m));
        funcTmp = func(Xf, newX);
        set.insert(setElemOneVar(newX, funcTmp));
        if (res.z > funcTmp) {
            res.y = newX;
            res.z = funcTmp;
        }
        if (maxRiter->x - maxPrevRiter->x <= _eps)
            timeToStop = true;
    }
```

```
return res;
}

bool compareResults(const resultTwoVar& a, const resultTwoVar& b, const double& eps) {
    bool equals = false;
        if (std::abs(static_cast<double>(a.x - b.x)) <= eps)
            if (std::abs(static_cast<double>(a.y - b.y)) <= eps)
                if (std::abs(static_cast<double>(a.z - b.z)) <= eps)
                    equals = true;
    return equals;
}
```

main.cpp

```
// Copyright 2019 Nazarov Vladislav
#define _USE_MATH_DEFINES

#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include "./global_optimization.h"

TEST(Global_Optimization_MPI, Test_First_Function) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    double(*fptr)(double, double) = f1;
    resultTwoVar res = solveTwoVar(-5, 5, -5, 5, fptr);
    if (rank == 0) {
        resultTwoVar resEq = solveTwoVarSequential(-5, 5, -5, 5, fptr);
        ASSERT_TRUE(compareResults(res, resEq, 0.1));
    }
}

TEST(Global_Optimization_MPI, Test_Second_Function) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    double(*fptr)(double, double) = f2;
    resultTwoVar res = solveTwoVar(-0.9, 1, -1, 1, fptr);
    if (rank == 0) {
        resultTwoVar expectedRes = {0, 0, 4};
        ASSERT_TRUE(compareResults(res, expectedRes, 0.1));
    }
}

TEST(Global_Optimization_MPI, Test_Third_Function) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    double(*fptr)(double, double) = f3;
    resultTwoVar res = solveTwoVar(0, 2, 0.3, 1, fptr);
    if (rank == 0) {
        resultTwoVar expectedRes = {1, 0.5, 4};
        ASSERT_TRUE(compareResults(res, expectedRes, 0.1));
    }
}

TEST(Global_Optimization_MPI, Test_Fourth_Function) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    double(*fptr)(double, double) = f4;
    resultTwoVar res = solveTwoVar(0.1, 5, -5, 5, fptr);
    if (rank == 0) {
        resultTwoVar expectedRes = {5, -5, -136.18};
        ASSERT_TRUE(compareResults(res, expectedRes, 0.1));
    }
}

TEST(Global_Optimization_MPI, Test_Fifth_Function) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    double(*fptr)(double, double) = f5;
    resultTwoVar res = solveTwoVar(1, 3, 1, 2, fptr);
```

```
    if (rank == 0) {
        resultTwoVar expectedRes = {2, 1.25, -1.5};
        ASSERT_TRUE(compareResults(res, expectedRes, 0.1));
    }
}

TEST(Global_Optimization_MPI, Test_Sixth_Function) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    double(*fptr)(double, double) = f6;
    resultTwoVar res = solveTwoVar(1, 5, 0.5, 4, fptr);
    if (rank == 0) {
        resultTwoVar expectedRes = {3*M_PI_2, M_PI, -3};
        ASSERT_TRUE(compareResults(res, expectedRes, 0.1));
    }
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    MPI_Init(&argc, &argv);

    ::testing::AddGlobalTestEnvironment(new GTestMPIListener::MPIEnvironment);
    ::testing::TestEventListeners& listeners =
        ::testing::UnitTest::GetInstance()->listeners();

    listeners.Release(listeners.default_result_printer());
    listeners.Release(listeners.default_xml_generator());

    listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);
    return RUN_ALL_TESTS();
}
```