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

«Алгоритм глобального поиска Стронгина для одномерных задач оптимизации»

Выполнил:
студент группы 3821Б1ПМоп3
Иванченко А. М.

Проверил:
младший научный сотрудник
Нестеров А. Ю.

Нижний Новгород
2023

Содержание

Введение

Многие задачи принятия оптимальных решений, возникающие в различных сферах человеческой деятельности, могут быть сформулированы как задачи оптимизации. При быстро возрастающей сложности оптимизируемых объектов совершенно естественным является и усложнение их математических моделей, что, как следствие, значительно затрудняет поиск оптимальной комбинации параметров. Очень часто не представляется возможным найти такую комбинацию аналитически и возникает необходимость построения численных методов для ее поиска.

Проблема численного решения задач оптимизации, в свою очередь, может быть сопряжена со значительными трудностями. Во многом они связаны с размерностью и видом оптимизируемой целевой функции. При этом численные методы глобальной оптимизации существенно отличаются от стандартных локальных методов поиска, которые часто неспособны найти глобальное (т. е. абсолютно лучшее) решение рассматриваемых задач в силу многоэкстремальности целевой функции. Локальные методы, как правило, оказываются не в состоянии покинуть зоны притяжения локальных оптимумов и, соответственно, упускают глобальный оптимум. Использование же найденных локальных решений может оказаться недостаточным, поскольку глобальное решение может дать существенный выигрыш по сравнению с локальными.

Одним из методов решения одномерных задач глобальной оптимизации является метод Стронгина. Целью данной работы является реализация метода глобального поиска Стронгина и его адаптация под паралльные вычисления с использованием MPI.

Постановка задачи

Постановка задачи глобального поиска

Метод Стронгина — метод решения одномерных задач условной липшицевой оптимизации. Позволяет находить глобально оптимальное решение в задачах с ограничениями неравенствами при условии, что целевая функция задачи и левые части неравенств удовлетворяют условию Липшица в области поиска.

Требуется найти точку $x^* \in [a;\ b]$ такую, что $(x^*) = \min\{f(x) \colon x \in [a;\ b],\ g_j(x) \leq 0,\ 1 \leq j \leq m\}$. Предполагается, что функции f(x) и $g_j(x),\ j = \overline{1,\ m}$ удовлетворяют условию Липшица на отрезке $[a;\ b]$. Обозначим $g_{m+1}(x) = f(x)$, тогда для $j = \overline{1,\ m+1}$ выполняются следующие неравенства:

$|g_j(x + \Delta x) - g_j(x)| \leq L_j \Delta x,\ a \leq +\Delta x \leq b$,

где $L_j \geq 0$ — константы Липшица.

Техническое задание

В рамках данной работы необходимо выполнить следующие задачи:

1. Реализовать метод глопального поиска Стронгина на языке программирования C++ (последовательная реализация)
2. Добавить параллельную реализацию метода на языке программирования C++ с приминением технологии MPI
3. Сравнить результаты и время работы разных реализаций алгоритма на различных наборах данных

Реализция

Описание алгоритма

Пусть $x^0 = a,\ x^1 = b$. Индексы концевых точек считаются нулевыми, а значения z в них не определены. Первое испытание осуществляется в точке $x^3 = (a+b)/2$. Выбор точки $x^{k+1},\ k \geq 3$ любого последующего испытания определяется следующими правилами:

- Перенумеровать точки $x^0,\ \ldots,\ x^k$ k предшествующих испытаний нижними индексами в порядке увеличения значений координаты:
  $a = x_0 < \ldots < x_i < \ldots < x_k = b$
  и сопоставить им значения $z_i = g_\nu(x_i),\ \nu = \nu(x_i),\ i = \overline{1,\ k}$.

- Для каждого целого числа $\nu,\ 1 \leq \nu \leq m+1$ определить соответствующее ему множество I_ν нижних индексов точек, в которых вычислялись значения функций $g_\nu(x)$:
  $I_\nu = \{i\colon \nu(x_i) = \nu,\ 1 \leq i \leq k\},\ 1 \leq \nu \leq m+1$.
  Также определить максимальное значение индекса $M = \max\{\nu(x_i),\ 1 \leq i \leq k\}$.

- Вычислить текущие оценки для неизвестных констант Липшица:
  $\mu_\nu = \max\{|g_\nu(x_i) - g_\nu(x_j)|/(x_i - x_j)\colon i,\ j \in I_\nu,\ i > j\}$.
  Если множество I_ν содержит менее двух элементов или если значение μ_ν оказывается равным нулю, то
  принять $\mu_\nu = 1$.

- Для всех непустых множеств $I_\nu,\ \nu = \overline{1,\ M}$ вычислить оценки
  $$z^*_\nu = \begin{cases} \min\{g_\nu(x_i)\colon x_i \in I_\nu\} & \nu = M, \\ -\varepsilon_\nu & \nu < M, \end{cases}$$
  где вектор с неотрицательными координатами $\varepsilon_R = (\varepsilon_1,\ \ldots,\ \varepsilon_m)$ называется *вектором резервов*.

- Для каждого интервала $(x_{i-1};\ x_i),\ 1 \leq i \leq k$ вычислить характеристику
  $$R(i) = \begin{cases} \Delta_i + \frac{(z_i - z_{i-1})^2}{(r_\nu\mu_\nu)^2\Delta_i} - 2\frac{z_i + z_{i-1} - 2z^*_\nu}{r_\nu\mu_\nu} & \nu = \nu(x_i) = \nu(x_{i-1}), \\ 2\Delta_i - 4\frac{z_{i-1} - z^*_\nu}{r_\nu\mu_\nu} & \nu = \nu(x_{i-1}) > \nu(x_i), \\ 2\Delta_i - 4\frac{z_i - z^*_\nu}{r_\nu\mu_\nu} & \nu = \nu(x_i) > \nu(x_{i-1}), \end{cases}$$
  где $\Delta_i = x_i - x_{i-1}$.
  Величины $r_\nu > 1,\ \nu = \overline{1,\ m}$ являются параметрами алгоритма. От них зависят произведения $r_\nu\mu_\nu$, используемые при вычислении характеристик в качестве оценок неизвестных констант Липшица.

- Определить интервал $(x_{t-1};\ x_t)$, которому соответствует максимальная характеристика $R(t) = \max\{R(i),\ 1 \le i \le k\}$.

- Провести очередное испытание в середине интервала $(x_{t-1};\ x_t)$, если индексы его концевых точек не совпадают:
  $x^{k+1} = \frac{1}{2}(x_t + x_{t-1})$.
  В противном случае провести испытание в точке
  $x^{k+1} = \frac{1}{2}(x_t + x_{t-1}) - \frac{z_t - z_{t-1}}{2r_\nu \mu_\nu},\ \nu = \nu(x_t) = \nu(x_{t-1})$,
  увеличить k на 1.

- Если $x_t - x_{t-1} < \varepsilon$ ($\varepsilon > 0$ — заданная точность метода), то прекратить выполнение алгоритма, иначе перейти на шаг 1.

Описание схемы распараллеливания

Алгоритм Стронгина относится к классу одношагово-оптимальных, следовательно при параллельной реализации алгоритм следует модифицировать таким образом, чтобы полученный алгоритм тоже принадлежал данному классу.

При параллельной реализации мы распределяем вычислительную нагрузку при подсчете функции характеристик интервалов, а так же оценок константы Липшица для интервалов. При этом сам алгоритм аналогичен исходному. Изменяется логика работы исключительно внутри итерации. В результате работы алгоритмы будет построено тоже множество точек, что и при последовательной реализации, что позволяет сохранить структуру алгоритма.

Message Passing Interface (MPI, интерфейс передачи сообщений) — программный интерфейс для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Разработан для запуска процессов, совместно выполняющих некоторую задачу и рассчитан на системы с разделенной памятью. Следовательно, основными моментами, на которые следует обращать внимание при написании алгоритмов параллельных MPI программ является проблема обмена данными. Каждый процесс хранит свою локальную копию всех данных. Возникает проблема оптимизации количества пересылок данных между процессами(операция пересылки по сети — является трудоемкой операцией).

На каждом вычислительном узле должна храниться своя локальная копия массива точек проведения испытаний. В результате после каждого вычисления следующей точки испытание необходима рассылка данной точки всем процессорам.

Программная реализация

Реализация метода глобального поиска находистя в заголовочном файле *global_search_strongin.h* и файле исходного кода *global_search_strongin.c*. Проверка тестов и измерение времени выполнения происходит в *main.cpp*.

Функция searchSequential

```
double searchSequential(double (*f)(double), double x0, double x1, double
    eps)
```

Поиск глобального минимума функции $f(x)$ на отрезке $[x_0, x_1]$. Используется метод Стронгина, выполняемый последовательно одним процессом.
Входные данные:

- f – целевая функция
- x0 – левая граница отрезка
- x1 – правая граница отрезка
- eps – точность выхода на минимум

Возвращает: координату минимума.

Функция searchParallel

```
double searchParallel(double (*f)(double), double x0, double x1, double
    eps)
```

Поиск глобального минимума функции $f(x)$ на отрезке $[x_0, x_1]$. Используется метод Стронгина, выполняемый параллельно на разных процессах с использованием технологии MPI.
Входные данные:

- f – целевая функция
- x0 – левая граница отрезка
- x1 – правая граница отрезка
- eps – точность выхода на минимум

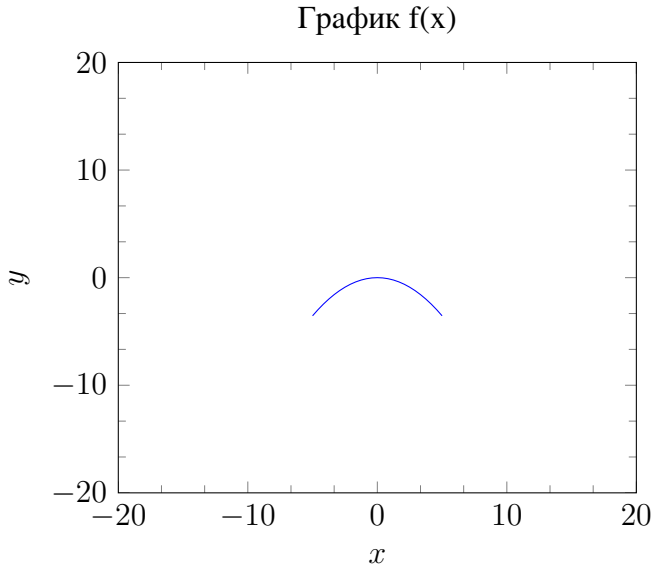
Возвращает: координату минимума.

Тестирование и проведение экспериментов

Подтверждение корректности

Для подтверждения корректности работы программы были написаны тесты с использованием Google Test. Пять тестов проверяют корректность последовательной реализации метода на простой задаче, решение которой легко найти аналитически. Остальные тесты сравнивают работу последовательного алгоритма с работой параллельного алгоритма на случайных наборах данных.

Проверка корректности последовательной реализации метода была проведена для функции
$f(x) = 4.176e - 6 * x^4 - 0.1419x^2$.
Полином был построен специальным образом, так, чтобы он достигал минимума в точках $x_1 = -130$,$x_2 = 130$. При этом функция на отрезке$[-200, 200]$ принимает не слишком большие значения, так что при проведении опытов задача хорошо масштабируется.



Сравнение параллельного и последовательного алгоритмов производится на той же самой целевой функции, но на случайно выбранных отрезках разной ненулевой длины.

Результаты экспериментов

Для проведения экспериментов по вычислению эффективности работы разных реализаций программы использовалась система со следующей конфигурацией:

- Процессор: Intel Core i3-10105F, ядер: 4, потоков: 8;
- Оперативная память: 16 ГБ
- Операционная система: Ubuntu 22.04.

Эксперимент ставился следующим образом: для одной и той же функции с одинаковым параметром точности eps проводился глобальный поиск на разных отрезках. Для каждого эксперимента создавалось 4 процесса. Измерялось время выполнения последовательного алгоритма, параллельного алгоритма, а затем вычислялось ускорение (как отношение времени работы параллельного алгоритма к времени последовательного).

Результаты экспериментов:

Отрезок	Ускорение (раз)
$[-1, 1]$	0.241361
$[-10, 10]$	2.13017
$[-50, 50]$	2.44291
$[-200, 200]$	3.28401

Таблица 1: Результаты экспериментов: ускорение при использовании параллельного алгоритма на разных отрезках

Заметим, что на малых отрезках распараллеливание не дает преимущества из-за накладных расходов. Однако на больших отрезках ускорение приближается к 4 (число процессов). Такое поведение является нормальным.

Заключение

Таким образом, в рамках данной лабораторной работы были разработаны последовательный и параллельный алгоритмы глобального поиска методом Стронгина. Проведенные тесты показали корректность реализованной программы, а проведенные эксперименты доказали эффективность распараллеливания этого алгоритма.

Литература

1. Баркалов К. А., Стронгин Р. Г. Метод глобальной оптимизации с адаптивным порядком проверки ограничений //Журнал вычислительной математики и математической физики. – 2002. – Т. 42. – №. 9. – С. 1338-1350.

2. Городецкий С. Ю., Гришагин В. А. Нелинейное программирование и многоэкстремальная оптимизация //Н. Новгород: Изд-во ННГУ. – 2007. – С. 357.

3. Стронгин Р. Г. Численные методы в многоэкстремальных задачах. – 1978.

4. Маркин Д. Л., Стронгин Р. Г. Метод решения многоэкстремальных задач с невыпуклыми ограничениями, использующий априорную информацию об оценках оптимума //Журнал вычислительной математики и математической физики. – 1987. – Т. 27. – №. 1. – С. 52-62.

5. Корняков К. В. и др. Инструменты параллельного программирования в системах с общей. – 2010.

Приложение

global_search_strongin.h

```
// Copyright 2023 Ivanchenko Aleksei
#ifndef
    TASKS_TASK_3_IVANCHENKO_A_GLOBAL_SEARCH_STRONGIN_GLOBAL_SEARCH_STRONGIN_H_
#define
    TASKS_TASK_3_IVANCHENKO_A_GLOBAL_SEARCH_STRONGIN_GLOBAL_SEARCH_STRONGIN_H_

#include <vector>

double searchSequential(double x0, double x1, double eps);
double searchParallel(double x0, double x1, double eps);

#endif //
    TASKS_TASK_3_IVANCHENKO_A_GLOBAL_SEARCH_STRONGIN_GLOBAL_SEARCH_STRONGIN_H_
```

global_search_strongin.cpp

```
// Copyright 2023 Ivanchenko Aleksei
#include <algorithm>
#include <random>
#include <boost/mpi/communicator.hpp>
#include <boost/mpi/collectives.hpp>
#include
    "task_3/ivanchenko_a_global_search_strongin/global_search_strongin.h"

double f(double x) {
    return 4.176e-6 * x*x*x*x - 0.1419*x*x;
}

double searchSequential(double x0, double x1, double eps) {
    std::vector<double> x;
    std::vector<double> y;
    double M = 0.0;
    double m;
    double R = 0.0;
    int interval = 0;
    if (x1 > x0) {
        x.push_back(x0);
        x.push_back(x1);
    } else {
        x.push_back(x1);
        x.push_back(x0);
    }

    while (true) {
        for (size_t i = 0ull; i < static_cast<int>(x.size()); i++) {
            y.push_back(f(x[i]));
        }
        for (size_t i = 0ull; i < static_cast<int>(x.size()) - 1ull; i++) {
            double lipsh = std::abs((y[i + 1ull] - y[i]) / (x[i + 1ull] -
                x[i]));
            if (lipsh > M) {
                M = lipsh;
                m = lipsh + lipsh;
                double tempR = m * (x[i + 1] - x[i]) +
```

```
                    pow((y[i + 1] - y[i]), 2) / (m * (x[i + 1] - x[i])) -
                    2 * (y[i + 1] + y[i]);
                if (tempR > R) {
                    R = tempR;
                    interval = i;
                }
            }
        }
        if (x[interval + 1] - x[interval] <= eps) {
            return y[interval + 1];
        }
        double newX = 0.0;
        newX = (x[interval + 1] - x[interval]) / 2 + x[interval] +
               (y[interval + 1] - y[interval]) / (2 * m);
        x.push_back(newX);
        sort(x.begin(), x.end());
        y.clear();
    }
}
double searchParallel(double x0, double x1, double eps) {
    int size, rank;
    MPI_Status status;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        std::vector<double> x;
        x.push_back(x0);
        x.push_back(x1);
        while (true) {
            sort(x.begin(), x.end());
            int part = static_cast<int>(x.size() - 1) / size;
            int remain = static_cast<int>(x.size() - 1) % size;
            for (int i = 1; i < size; ++i) {
                MPI_Send(x.data() + remain + i * part, 1, MPI_DOUBLE, i, 0,
                       MPI_COMM_WORLD);
            }
            double M = 0, m, lipsh;
            for (int i = 0; i < part + remain; ++i) {
                lipsh = std::abs((f(x[i + 1]) - f(x[i])) / (x[i + 1] -
                    x[i]));
                if (lipsh > M) {
```

```
                M = lipsh;
                m = 2 * lipsh;
            }
        }
        if (part > 0) {
            for (int i = 1; i < size; i++) {
                MPI_Recv(&lipsh, 1, MPI_DOUBLE, i, MPI_ANY_TAG,
                    MPI_COMM_WORLD,
                     &status);
                if (lipsh > M) {
                    M = lipsh;
                    m = 2 * lipsh;
                }
            }
        }
        int interval = 0;
        double tempR;
        double R = 0.0;
        for (int i = 0; i < static_cast<int>(x.size()) - 1; i++) {
            tempR = m * (x[i + 1] - x[i]) +
                  pow((f(x[i + 1]) - f(x[i])), 2) / (m * (x[i + 1] -
                      x[i])) -
                  2 * (f(x[i + 1]) + f(x[i]));
            if (tempR > R) {
                R = tempR;
                interval = i;
            }
        }

        if (x[interval + 1] - x[interval] <= eps) {
            for (int i = 1; i < size; ++i)
            MPI_Send(x.data(), 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD);
            return x[interval + 1];
        }

        double newX = (x[interval] + x[interval + 1]) / 2 -
                    (f(x[interval + 1]) - f(x[interval])) / (m + m);
        x.push_back(newX);
    }
} else {
    int part = 0;
```

```
        while (true) {
            MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, MPI_DOUBLE, &part);
            std::vector<double> x(part + 1);
            MPI_Recv(x.data(), 1, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
                    &status);
            if (status.MPI_TAG == 1)
              return 0;
            double M = 0, m = 1.0, lipsh;
            if (part != 0) {
                for (int i = 0; i < static_cast<int>(x.size()) - 1; ++i) {
                    lipsh = (std::abs(f(x[i + 1]) - f(x[i]))) / (x[i + 1] -
                        x[i]);
                    if (lipsh > M) {
                        M = lipsh;
                        m = M + M;
                    }
                }
            }
            MPI_Send(&m, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        }
        return 0;
    }
}
```