Федеральное государственное автономное образовательное учреждение
высшего образования «Национальный исследовательский университет
«Высшая школа экономики»

Факультет компьютерных наук
Основная образовательная программа
Прикладная математика и информатика

КУРСОВАЯ РАБОТА

Исследовательский проект на тему "Разработка высокопроизводительных методов глобальной оптимизации"

Выполнил студент группы 182, 3 курса,
Иванов Семен Вадимович

Руководитель КР:
профессор Посыпкин Михаил Анатольевич

Москва 2021

Содержание

Абстракт

В проекте планируется изучение методов глобальной оптимизации, их исследование, развитие и эффективная реализация на современных высокопроизводительных вычислительных системах. Рассматривается безызбыточная стратегия разбиения для диагонального подхода для Липшицевой оптимизации. Предполагается реализация на языке C++. Для реализации планируется использовать современные средства многопоточного программирования для CPU.

Ключевые слова: глобальная оптимизация, Липшицева оптимизация, многопоточность, диагональный подход.

Abstract

The project plans to study the methods of global optimization, develop their effective implementation on modern high-performance computing systems. A non-redundant partitioning strategy for the diagonal approach for Lipschitz optimization is considered. Method will be implemented in C++. It is planned to use modern multi-threaded programming tools for the CPU.

Keywords: global optimization, Lipschitz optimization, multithreading, diagonal approach.

Введение

Множество прикладных задач может быть сформулировано в виде задачи оптимизации. В связи с этим возникает задача глобальной оптимизации: нахождение глобального оптимума функции. Время нахождения данного оптимума в прикладных задачах стоит весьма остро, поэтому возникает необходимость в развитии и исследовании эффективных по времени алгоритмов глобальной оптимизации.

Можно предположить, что разумнее было бы искать численное решение и всегда получать точный оптимум. Однако из-за возрастающей сложности оптимизируемых процессов, их численное решение или невозможно, так как функция подается в виде черного ящика и ее исследование и анализ может занимать продолжительное время, или вычислительно нецелесообразно из-за слишком большого числа оптимизируемых параметров, то есть из-за высокой размерности (например, как вычисление точных коэффициентов в линейной регрессии).

Отметим также, что методы локальной оптимизации не всегда могут подходить для решения данной задачи, так как они зачастую не в состоянии покинуть локальные оптимумы, которые могут давать значительно меньший эффект в сравнении с глобальным решением. Поэтому необходима дополнительная разработка методов глобальной оптимизации, позволяющих гарантированно достичь глобального оптимума за наименьшее число вычислений значения функции. Методы локальной оптимизации разумно использовать, когда глобальный метод уже начинает сходится, гиперинтервал сильно уменьшается и нужен более тонкий подбор.

В работе планируется рассмотреть диагональный подход с безызбыточной стратегией разбиения для глобальной оптимизации многомерных функций и разработать его многопоточную версию. В рамках данного подхода предполагается, что вычисление функции – трудозатратная операция и число ее вызовов необходимо оптимизировать.

Задача, решаемая в проекте

Чтобы разрабатываемые методы для решения задачи были более эффективными, на задачу, как правило, накладывается ряд априорных предположений о ее характере. Первое важное свойство оптимизируемой функции это ее вычислительная трудозатратность. Также предполагается, что задача имеет вид безусловной Липшицевой глобальной оптимизации с неизвестной константой Липшица и подается в виде черного ящика.

Сформулируем задачу глобальной оптимизации на гиперинтервале[1] (для $f : \mathbb{R}^N \to \mathbb{R}$):

$$f^* = f(x^*) = \min_{x \in D} f(x), \text{ где} \tag{1}$$

$$D = \left\{ x \in \mathbb{R}^N : a_j \leq x_j \leq b_j, 1 \leq j \leq N \right\} \tag{2}$$

Иными словами, требуется найти пару $(x^*, f(x^*))$ в гиперкубе D (с некоторой точностью).

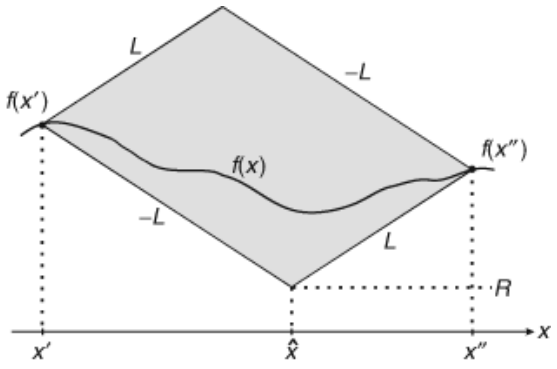

Рис. 1: Функция, удовлетворяющая условию Липшица с константой L на интервале $[x'; x'']$

Теперь введем ограничение на целевую функцию $f(x)$. В данной работе рассматриваются Липшицевые функции. Функция называется Липшицевой, если она удовлетворяет условию Липшица:

$$|f(x') - f(x'')| \leq L \left\| x' - x'' \right\|, \; x', x'' \in D, \; L > 0, \tag{3}$$

где L называется константой Липшица. Считается, что она заранее неизвестна.

Также заметим, что вычислительная трудозатратность оптимизиру-

емой функции слегка сдвигает баланс с точки зрения алгоритмов глобальной оптимизации, теперь лишние вычисления значений функции являются нежелательными и метод необходимо оптимизировать и с точки зрения числа вызовов функции $f(x)$.

Именно для такой постановки был предложен диагональный подход Липшицевой глобальной оптимизации с безызбыточной стратегией разбиения[1]. На Рис. 2 можно увидеть сравнение данного метода с двумя другими стандартными для разных задач. Далее в работе этот подход будет рассмотрен, описана и уточнена его эффективная последовательная реализация, предложен подход к многопоточной реализации и приведены некоторые эксперименты, демонстрирующие его работу.

n	ε	Class	DIRECT	BISECTION	SMOOTHD	MULTK
2	10^{-4}	Simple	198.89	432.75	151.11	74.75
2	10^{-4}	Hard	1,063.78	707.03	404.79	162.11
3	10^{-6}	Simple	1,117.70	3,369.76	1,011.00	783.49
3	10^{-6}	Hard	$\gg$6,322.65	4,934.85	1,756.18	618.32
4	10^{-6}	Simple	$\gg$11,282.89	4,061.15	4,598.97	3,512.92
4	10^{-6}	Hard	$\gg$29,540.12	>59,581.96	7,276.23	6,127.09
5	10^{-7}	Simple	>6,956.97	>40,772.45	4,281.42	3,583.20
5	10^{-7}	Hard	$\gg$72,221.24	>50,223.86	33,246.18	19,688.68

Рис. 2: Разные методы глобальной оптимизации по числу вызовов $f(x)$ (последние 2 основаны на диагональной схеме)

Обзор литературы

[1, 2] — монографии, в которых собраны основные одномерные методы глобальной оптимизации, а далее приведен диагональный поход и его вариации/улучшения, позволяющие обощить плоские решения на более высокие размерности, также присутствуют оценки и доказательства сходимости метода. В ней описан безызбыточный диагональный способ разбиения. Строго говоря, эта книга представляет из себя компиляцию статей, развивающих тему диагонального подхода.

[3] – статья, которая была взята в качестве примера и ориентира разработки и тестирования методов.

[4] – набор стандартных задач многомерной глобальной оптимизации, из которого планируется выбрать некоторое подмножество для тестирования метода.

[5] – Документация Intel Threading Building Blocks, в которой реализованы многопоточные примитивы и на основании которой планируется разработка многопоточной части.

[6] – Документация C++, языка, на котором разрабатывалась программная реализация.

План работы

1 Освоить 1-d случай

Разобрать способы решения задачи при $N = 1$. Планируется изучить данные методы на основе [1], чтобы иметь представление об области и существующих подходах.

2 Изучить диагональные схемы и безызбыточную стратегию разбиения

В рамках данного пункта необходимо изучить подход, распараллеливание которого планируется разрабатывать [1, 2]

3 Эффективная последовательная реализация

Разработать и реализовать эффективную последовательную версию интересующего алгоритма.

4 Распараллеливание

Планируется доработать полученную последовательную схему и сделать ее более пригодной для многопоточного исполнения.

5 Исследование эффективности разработанного алгоритма

Провести серию тестов на наборе задач. И исследовать полученные результаты с точки зрения эффективности распараллеливания и сохранения безызбыточности.

Диагональный подход с безызбыточной стратегией разбиения

Диагональный подход

Этот подход был предложен и развит Сергеевым Я. Д. и Квасовым Д. Е. Более детально с ним (в том числе с теоретическими выводами) можно ознакомиться в книге [1]. Опишем общюю схему данного подхода (рассматривается задача оптимизации функции $f : \mathbb{R}^N \to \mathbb{R}$):

```
1 while not Tasks.StoppingCriterion():
2     task = Tasks.PopBestTask()
3     taskDivided = DivideTask(task)
4     for task in taskDivided:
5         Tasks.PushTask(task)
6     Tasks.Recalc()
```

Листинг 1: Общая схема диагонального подхода

Прокомментируем. Здесь $Tasks$ – это список подзадач, то есть в нашем случае список гиперкубов вида (2). Каждая подзадача характеризуется двумя точками a и b (и соответственно значениями в этих точках $f(a)$ и $f(b)$) на какой-то из главных диагоналей гиперкуба (например, в квадрате есть 2 главные диагонали, в кубе – 4), а также некоторым критерием R, который вычисляется на основе этих точек и оценки константы Липшица и характеризует желательность данного фрагмента к разбиению (иными словами, чем он больше, тем раньше он будет исследован и разбит на меньшие). Из определения (3) ясно, что константу Липшица можно оценивать снизу как $\frac{|f(a)-f(b)|}{\|a-b\|} \leq L$, будем брать максимум по таким оценкам и обозначим ее через $\hat{L}$.

Алгоритм работает пока не выполнится некоторый критерий останова (строчка 1). В [1] предлагается разбивать до тех пор, пока выполняется $\|a - b\| > \varepsilon \|A - B\|$, где A и B – диагональные точки исходного гиперкуба, а ε – заданный заранее гиперпараметр, характеризующий точность (со-

ответственно, чем он меньше, тем ближе алгоритм должен подобраться к глобальному оптимуму). Во второй строчке из списка достается подзадача с самым большим значением критерия R для дальнейшего разбиения. Далее, в 3 строчке, происходит разбиение данной подзадачи с вычислением всех необходимых значений, на выходе получаем список taskDivided, состоящий из подзадач, полученных после разбиения task. В 4-5 они добавляются в общий пул задач. В последней строчке производится пересчет каких-то параметров в связи с, например, изменением оценки константы Липшица, увеличением числа подзадач и так далее.

В качестве критерия R было решено взять информационно--статистический критерий, предложенный в [1]:

$$R = \mu \|a - b\| + \frac{(f(a) - f(b))^2}{\mu \|a - b\|} - 2(f(a) + f(b)), \tag{4}$$

где $\mu = (r + \frac{C}{k}) \max(\xi, \hat{L})$ – текущая оценка константы Липшица (r и C – гиперпараметры, k – номер итерации; ξ – малое число, чтобы оценка не занулялась). Отметим, что в качестве критерия можно взять любой другой критерий, например, из методов одномерной оптимизации.

Сразу опишем стратегию разбиения интервала, на которой будет основана безызбыточная стратегия. Предлагается разбивать гиперкуб двумя плоскостями, перпендикулярными наиболее длинной его стороне на 3 равные части. Для наглядности на Рис. 3 показано, как будет происхо-

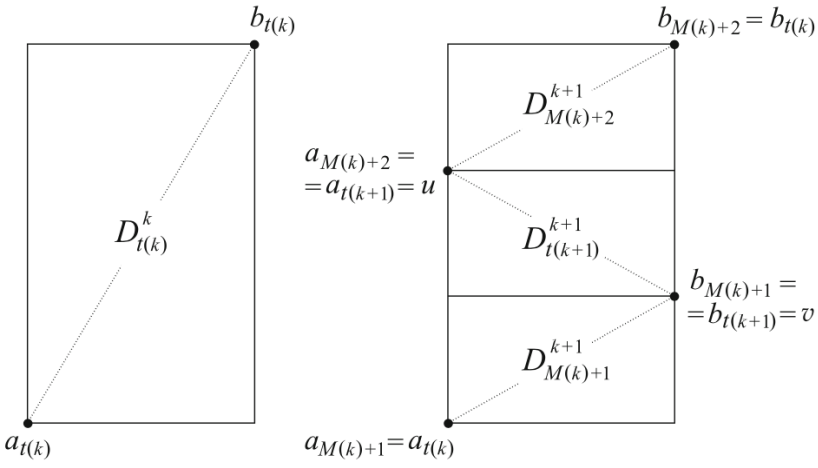

Рис. 3: Иллюстрация безызбыточной стратегии разбиения для функции двух переменых

дить разбиение в случае функции двух переменных. Вначале имеется одна подзадача с двумя точками. Дальше происходит разбиение по самой протяженной стороне и добавляются две новый точки u и v. В итоге мы получаем три новых гиперкуба, вычисляя дополнительно значение в двух точках. Также следует обратить внимание на то, какие точки после разбиения являются для подазадачи a, а какие b, это будет важно для дальнейшего алгоритма.

Безызбыточная стратегия разбиения

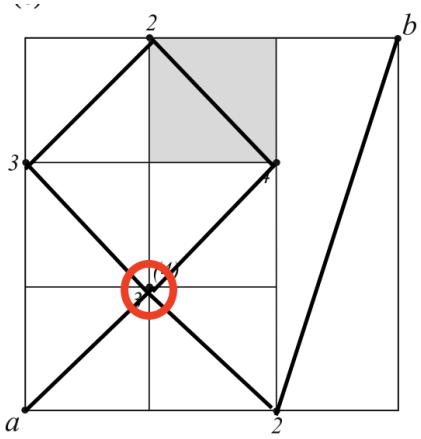

Рис. 4: Иллюстрация возможной избыточности стратегии разбиения

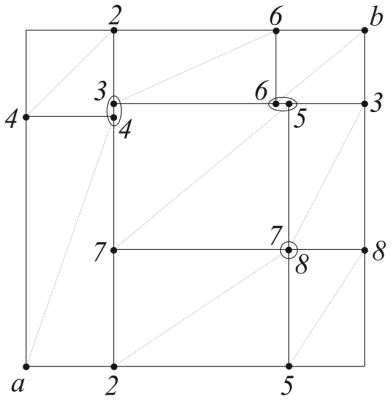

Рис. 5: Иллюстрация избыточности при неравномерном разбиении

Теперь опишем, как добиться безызбыточности описанной выше стратегии разбиения. Здесь безызбыточность понимается в стремлении не вычислять повторно значение функции. Ради наглядности снова обратимся к случаю функции двух переменных. На Рис. 4 можно пронаблюдать ситуацию, где после нескольких разбиений точка, выделенная красным цветом, присутствует в четырех фрагментах. Таким образом, если мы просто следуем стратегии, нам приходится вычислять значение функции в этой точке на 3 раза больше, чем могло бы быть. Если обощить на произвольную размерность N, то получается, что в худшем случае стратегия вычисляет функцию в 2^N раз больше, чем стоило, что при вычислительной затратности самой функции очень нежелательно.

Также отметим важный момент, что если бы мы заранее не выбрали

именно такую равномерную стратегию разбиения, то столкнулись бы с более неявной избыточностью, как на Рис. 5, где обведены точки, которые находятся близко друг другу и используются в разных прямоугольниках. Учитывая липшицевость функции и близость точек, эти точки с большой вероятностью слабо отличаются друг от друга. Иными словами, мы могли бы вычислить значение только в одной из них и считать, что во второй значение такое же и почти наверное бы ничего не потеряли с точки зрения сходимость и точности.

Теперь обозначим ключевые моменты, которые позволят достичь безызбыточности описанной стратегии. Основная идея, предложенная в [1], заключается в том, что мы можем разделить гиперкубы и точки на две разные сущности и построить из множества гиперкубов инъекцию в множество точек. Если такая инъекция возможна, то есть возможно раздельное хране информации о подзадачах и вычисленных точках и возможно соответствие из подзадач в соответствующие точки, то достигается безызбыточность, так как мы можем перед вычислением новой точки воспользоваться этим соотетствием и взять уже имеющееся значение, если такое есть.

Обсудим способ построения. Будем кодировать подзадачи N строчками, которые будут состоять из 0, 1 и 2. Положим, что исходная подзадача кодируется N строчками из одного нуля: $(0, \cdots, 0)$. Теперь допустим у нас есть код некоторой подзадачи $(s_1, \cdots, s_N)$ и мы хотим разбить ее по k-той координате на три меньшие подзадачи (описанным выше способом). Тогда у них будут соответственно коды $(s_1, \cdots, s_{k-1}, s_k 0, s_{k+1}, \cdots, s_N)$, $(s_1, \cdots, s_{k-1}, s_k 1, s_{k+1}, \cdots, s_N)$ и $(s_1, \cdots, s_{k-1}, s_k 2, s_{k+1}, \cdots, s_N)$. Иными словами, мы приписали к k-тому коду все три возможных символа. Здесь важно отметить момент, про который не удалось найти упоминание, что первый код соответствует подзадаче наиболее близкой к левой начальной точке A, второй центральной, а третий наиболее близкой к точке B.

Перейдем к построению соответствия в множество точек. Для простоты будем считать, что $A = (0, \cdots, 0)$, $B = (1, \cdots, 1)$, произвольный случай можно либо привести к этому, либо провести аналогичные рассуждения.

Ключевая идея заключается в том, чтобы каким-то образом преобразовать коды $(s_1, \cdots, s_N)$ к $(\hat{s}_1, \cdots, \hat{s}_N)$ и $(\overline{s}_1, \cdots, \overline{s}_N)$ и координаты двух точек подзадачи можно было бы вычислять по формуле $a_i = \sum_{j=1}^{N} \frac{\hat{s}_{ij}}{3^j}$ и аналогичной для b из кодов $(\overline{s}_1, \cdots, \overline{s}_N)$.

Для начала заметим, что если выбирается центральная подзадача, то ее диагональ отличается от двух других тем, что она зеркально отражена относительно координаты разбиения (см. Рис. 3). Но при этом, если мы повторно выбираем центральную задачу при разделении по той же координате, то ее положение относительно этой координаты меняется обратно. В связи с этим предлагается следующая схема. Если код i-той координаты имеет четное число 1, то a_i вычисляется по формуле выше для исходного кода $\sum_{j=1}^{N} \frac{s_{ij}}{3^j}$. Вычисление b_i следует проводить по той же формуле, но для кода s_i+1. Здесь следует объяснить, что запись $s_i + 1$ понимается не совсем как циклическое прибавление 1 к последнему символу, как это на первый взгляд было описано в работе [1] (то есть $0 \to 1 \to 2 \to 0$), а как прибавление единицы к s_i как к числу в троичной системе счисления (например, $0011222 + 1 = 0012000$). Если же код i-той координаты имеет нечетное число 1, то координаты a_i и b_i вычисляются по той же сумме, но теперь из кодов $s_i + 1$ и s_i соответственно. Таким образом, отображение из множества кодов подзадачи в множество кодов соответствующих точек построено и описан способ перевода кода точки в сами координаты этой точки.

Реализация

Эффективная последовательная реализация

Несмотря на то, что в задаче предполагается вычислительно затратная функция, эффективная реализация данного подхода все равно необходима, так как прямая реализация, когда мы, например, просто храним список кодов точек и подзадач и при помощи этих структур реализуем алгоритм, может оказаться слишком неэффективной и устанавливать чрезмерно высокий порог в сложности оптимизируемой функции, начиная с которой безызбыточный алгоритм давал бы какой-либо выигрыш в сравнении с классическими решениями.

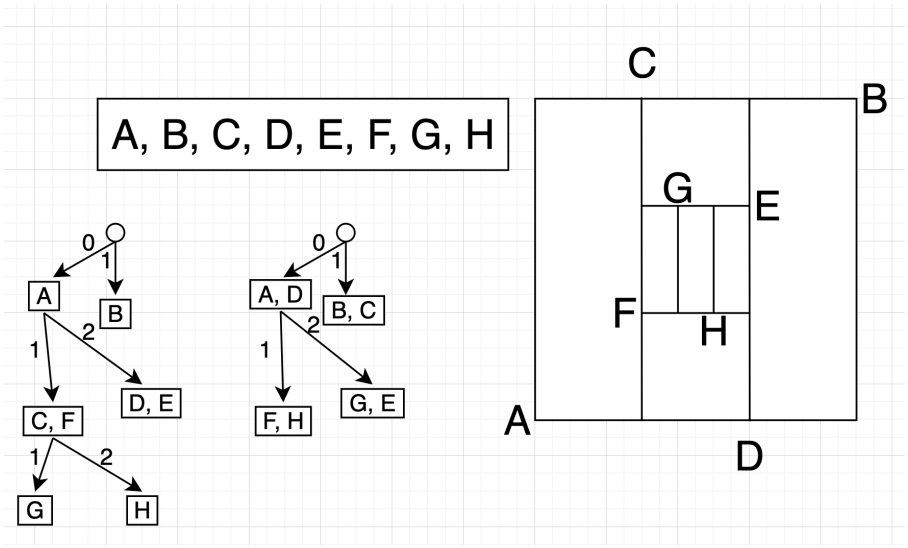

Рис. 6: Двумерный пример деревьев для хранения точек (слева дерево по оси X, справа по Y)

Реорганизуем базу точек. Заведем структуру дерева для кодов каждой координаты. Ребро дерева будет соответствовать добавлению в конец кода соответствующего символа (0, 1 или 2), в вершинах будут храниться точки, у которых код для данной координаты соответствет пути от корня до вершины. То есть, у нас есть N деревьев и, если мы возьмем N путей в каждом дереве до искомой точки, то получим коды по всем N размерностям. Теперь, чтобы проверить, присутствует ли точка в базе, достаточно обойти N деревьев за $O(\sum_i |s_i|)$ и проверить, есть ли в пересечении этих N вершин

индекс. Если есть, то такая точка присутствует и обратиться к ней можно по соответствующему индексу. Иначе необходимо добавить точку в деревья, создав новые вершины или дополнив списки уже существующих.

Также стоит отметить, что в кодах точек следует игнорировать нули в конце, то есть, например, считать, что $011200 = 01120 = 0112$. Без этого одинаковые точки, но сразным числом нулей в конце кодов считались бы разными и стратегия бы оказалась избыточной.

Теперь заметим, что можно избавиться от хранения достаточно объемной информации в виде кодов фрагментов и каждый раз при разбиении копировать и переводить ее в коды точек. Достаточно хранить указатели на вершины в деревьях для точек a и b. Опишем случай получения точек u и v, когда диагональ подзадачи не инвертирована по разбиваемой размерности (Рис. 3), то есть она была центральной по данной размерности нечетное число раз. Чтобы получить u необходимо взять вершины точки a, пройти нужное число нулевых ребер (см. предыдущий абзац) и перейти по ребру 2. Для v аналогично, только берется точка b и в конце перейти по ребру 1. Во втором случае, когда диагональ инвертирована, аналогично, только 1 и 2 меняются местами. Таким образом, мы получаем сложность в виде сложности на пересечение вершин, она линейна от размера вершин.

Параллельная реализация

В текущей реализации есть несколько мест из-за которых возможно возникнут проблемы при ее распараллеливании. Во-первых, так как порядок обработки подзадач зависит от их приоритета, то есть это поиск лучшего (best first search), существуют проблемы с адаптивным построением расписания обработки подзадач. Во-вторых, есть проблема с многопоточным достутом до описанной базы точек.

Предложим способ распараллеливания описанной базы точек. Введем структуру листа, которая позволяет многопоточно добавлять элементы

в конец и без lock-ов на структуру проитерироваться по всем ее элементам. В сущности эта структура достаточно проста: перед началом итерирования мы просто запоминаем размер уже точно созданных элементов и итерируемся фиксированное число раз; для добавления элементов мы блокируем (например, при помощи std::atomic или std::mutex) последнюю вершину листа и добавляем новый элемент. Однако стоит отметить, что у данной структуры есть минус, что во время итерации может быть добавлен какой-то новый элемент, по которому цикл не проитерируется. Впрочем для текущей постановки эта ситуация достаточно редкая и, как будет видно далее, не является критичной.

Данную структуру листа предлагается интегрировать вместо вектора индексов точек в вершины деревьев (см. Рис. 6). В таком случае, обращение к базе всегда будет бесплатным в плане блокировок других потоков, а запись, которая происходит значительно реже, будет блокировать небольшую область (только конец одного списка).

Теперь необходимо определиться с способом, который будет определять порядок исследования (шедулинг). В качестве базовой структуры было решено взять конкурентную приоритетную очередь реализации Intel Threading Building Blocks [5]. Также данную библиотеку было решено взять для организации работы потоков. Это приоритетная очередь с рядом эвристик, которые оптимизируют ее для использования несколькими потоками. Было выполнено небольшое тестирование и сравнение с std::priority_queue с использованием mutex-а. На 4 потоках при постоянном отношении числа добавлений и удалений, конкурентная очередь в отличие от стандартной реализации показала ускорение, примерно в 2 раза. Таким образом, в ней предлагается хранить подзадачи и их приоритеты и доставать следующую для обработки подзадачу. Здесь стоит отметить, что в какой-то момент, при достаточно большом числе потоков, одна очередь может оказаться узким местом в работе алгоритма, так как все потоки будут обращаться в одну и ту же структуру. В этом случае предлагается использовать несколько приоритетных очередей

и для обработки доставать подзадачу из случайной очереди и добавлять разбитые подзадачи тоже в случайные. В таком случае, у всех очередей должны быть примерно одинаковые по качеству (с точки зрения приоритета) задачи и исчезает риск появления узкого места.

Так как приоритет подзадачи зависит от константы Липшица, которая оценивается и может со временем сильно увеличиться, что скажется на порядке подзадач, его стоит иногда пересчитывать. Для этого предлагается ввести гиперпараметр на относительное изменение оценки константы Липшица с момента последнего пересчета приоритетов. Когда изменение вышло за выставленный порог, происходит пересчет всех приоритетов. Также возможно проводить пересчет критерия после определенного увеличения числа подзадач.

Эксперименты и выводы

Описанные выше подходы были реализованы и с ними можно ознакомиться в [7].

Было решено взять набор из стандартных многомерных оптимизационных задача [4] и замерить на них, насколько хорошо получилось распараллелить базу точек и оценить эффективность порядка обработки подзадач.

Здесь стоит еще раз отметить, что изначально этот метод предлагался для вычислительно трудозатратных функций, в связи с чем и организуется база точек и специальная схема разбиения. А так как такие функции, как правило, имеют более прикладной характер, было решено взять набор формульных стандартных многоэкстремальных многомерных задач [4] и провести анализ алгоритма по следующим параметрам.

Измерялось среднее ускорение времени исполнения задачи, то есть отношение времени работы многопоточной версии к последовательной, а также процентное изменение числа вычислений функции в точке при переходе от последовательной к многопоточной. Таким образом, по первому параметру

можно будет судить об эффективности распараллеливания базы и шедулинге подзадач. По второму можно будет понять, насколько сильно изменяется избыточность в многопоточной версии, а также, сильно ли избыточность будет сказываться при многопоточном исполнении.

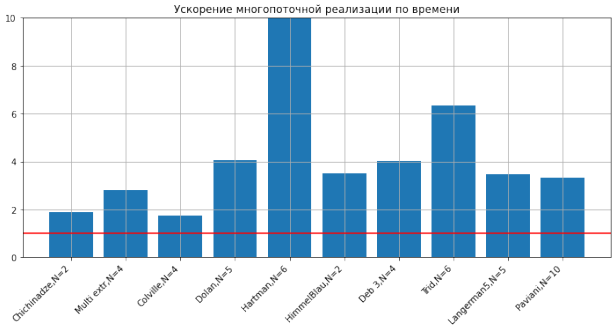

Рис. 7: Тестирование по времени

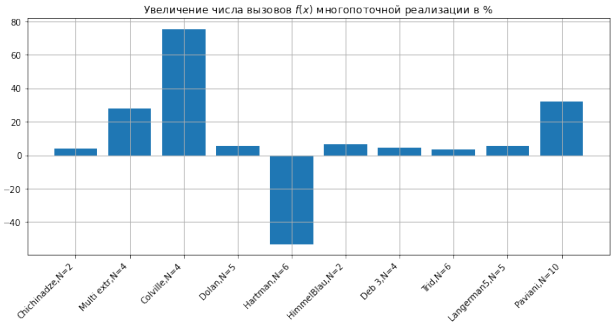

Рис. 8: Тестирование по числу вызовов $f(x)$

Тестирование проводилось на 5 потоках с одной приоритетной очередью. Успешным завершением алгоритма считалось определенное относительное приближение к глобальному оптимуму. [7]

Проанализируем полученные результаты. Во-первых, видно, что многопоточная версия везде дает ускорение по времени, в среднем в 3.5 раза. Это значит, что механика многопоточности и соответствующие структуры были реализованы достаточно разумно и их многопоточное использование имеет смысл. По второму же графику видно, что в среднем многопоточная реализация дает не сильное увеличение числа вызовов функции в точке, которое из-за своей незначительности не ухудшит результаты, так как теперь на один поток точно будет приходиться меньшее число вызовов функции. Не сильное увеличение числа вызовов функции можно также связать с достаточно жестким критерием для подзадачи, который в значительной степени зависит от размера подзадачи и не дает алгоритму идти сильно в глубь.

Таким образом, можно сделать вывод, что данный диагональный подход возможно распараллелить, сохранив его основное качество – безызбыточность.

В качестве возможных перспектив можно выделить несколько: про-

тестировать другие критерии для подзадач, внедрить схему с локальной настройкой [2] (стр. 90).

Список литературы

[1] Д.Е.Квасов Я.Д.Сергеев. *Диагональные методы глобальной оптимизации*. Москва, Нижний Новгород: ФИЗМАТЛИТ, 2008. ISBN: 9785922110327.

[2] Dmitri E. Kvasov Yaroslav D. Sergeyev. *Deterministic global optimization : an introduction to the diagonal approach*. New York, NY: Springer, 2017. ISBN: 9781493971978.

[3] М.А. Посыпкин А.Ю. Горчаков. *СРАВНЕНИЕ ВАРИАНТОВ МНОГОПОТОЧНОЙ РЕАЛИЗАЦИИ МЕТОДА ВЕТВЕЙ И ГРАНИЦ ДЛЯ МНОГОЯДЕРНЫХ СИСТЕМ*. г. Москва, Россия: Федеральный исследовательский центр «Информатика и управление» Российской академии наук, 2018.

[4] Xin-She Yang Momin Jamil. *A literature survey of benchmark functions for global optimization problems*. Int. Journal of Mathematical Modelling и Numerical Optimisation, 2013. URL: https://arxiv.org/pdf/1308.4008.pdf.

[5] *Intel Threading Building Blocks*. URL: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onetbb.html#gs.14fdl2.

[6] *cppreference*. URL: https://en.cppreference.com/w/.

[7] *Реализация описанных подходов*. URL: https://github.com/defunator/diagonal-go.