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

Направление подготовки: «Прикладная математика и информатика»
Магистерская программа: «Системное программирование»

ОТЧЁТ

по научно-исследоваетльской практике

на тему:

Разработка эффективных структурх хранения данных для алгоритмов глобальной оптимизации

Выполнил: студент группы 381503м2
__________________ Совраов В.В.
Подпись
Научный руководитель:
доцент, к.ф.м.н.
__________________ Баркалов К.А.
Подпись

Нижний Новгород
2016

Содержание

1. Введение

Задачи нелинейной глобальной оптимизации встречаются в различных прикладных областях и традиционно считаются одними из самых трудоёмких среди оптимизационных задач. Их сложность экспоненциально растёт в зависимости от размерности пространства поиска, поэтому для решения существенно многомерных задач требуются суперкомпьютерные вычисления.

В настоящее время на кафедре МОиСТ активно ведётся разработка программной системы для глобальной оптимизации функций многих вещественных переменных ExaMin. Эта система включает в себя последние теоретические разработки, сделанные на кафедре в этой сфере, в том числе и блочную многошаговую схему редукции размерности [1]. Отличительной чертой ситемы является то, что, она может работать как на CPU, так на разных типах ускорителей вычислений с высокой степенью параллельности (XeonPhi, GPU Nvidia) [2; 3].

В данной работе будут описаны некеторые улучшения, внесённые в систему, и предварительные исследования, проведённые перед их внедрением.

2. Постановка задачи глобальной липшицевой оптимизации

Одна из постановок задачи глобальной оптимизации звучит следующим образом: найти глобальный минимум N-мерной функции $\phi(y)$ в гиперинтервале $D = \{y \in R^N : a_i \leqslant x_i \leqslant b_i, 1 \leqslant i \leqslant N\}$. Для построения оценки глобального минимума по конечному количеству вычислений значения функции требуется, чтобы $\phi(y)$ удовлетворяла условию Липшица.

$$\phi(y^*) = \min\{\phi(y) : y \in D\}$$

$$|\phi(y_1) - \phi(y_2)| \leqslant L\|y_1 - y_2\|, y_1, y_2 \in D, 0 < L < \infty$$

Классической схемой редукции размерности для алгоритмов глобальной оптимизации является использование разверток — кривых, заполняющих пространство [4].

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

Такое отображение позволяет свести задачу в многомерном пространстве к решению одномерной ценой ухудшения её свойств. В частности, одномерная функция $\phi(y(x))$ является не Липшицевой, а Гёльдеровой:

$$|\phi(y(x_1)) - \phi(y(x_2))| \leqslant H|x_1 - x_2|^{\frac{1}{N}}, x_1, x_2 \in [0, 1]$$

где константа Гельдера H связана с константой Липшица L соотношением

$$H = 4Ld\sqrt{N}, d = \max\{b_i - a_i : 1 \leqslant i \leqslant N\}$$

3. Алгоритм глобального поиска

Для дальнейшего изложения потребуется описание метода глобальной оптимизации, используемого в системе ExaMin. Многомерные задачи сводятся к одномерным с помощью различных схем редукции размерноти, поэтому можно рассматривать минимизацию одномерной функции $f(x), x \in [0,1]$, удовлетворяющей условию Гёльдера.

Рассматриваемый алгоритм решения данной задачи предполагает построение последовательности точек x_k, в которых вычисляются значения минимизируемой функции $z_k = f(x_k)$. Процесс вычисления значения функции (включающий в себя построение образа $y_k = y(x_k)$) будем называть испытанием, а пару (x_k, z_k) — результатом испытания. Множество пар $\{(x_k, z_k)\}, 1 \leqslant k \leqslant n$ составляет поисковую информацию, накопленную методом после проведения n шагов.

На первой итерации метода испытание проводится в произвольной внутренней точке x_1 интервала $[0;1]$. Пусть выполнено $k \geqslant 1$ итераций метода, в процессе которых были проведены испытания в k точках $x_i, 1 \leqslant i \leqslant k$. Тогда точка x^{k+1} поисковых испытаний следующей $(k+1)$-ой итерации определяются в соответствии с правилами:

Шаг 1. Перенумеровать точки множества $X_k = \{x^1, \dots, x^k\} \cup \{0\} \cup \{1\}$, которое включает в себя граничные точки интервала $[0,1]$, а также точки предшествующих испытаний, нижними индексами в порядке увеличения значений координаты, т.е.

$$0 = x_0 < x_1 < \dots < x_{k+1} = 1$$

Шаг 2. Полагая $z_i = f(x_i), 1 \leqslant i \leqslant k$, вычислить величины

$$\mu = \max_{1 \leqslant i \leqslant k} \frac{|z_i - z_{i-1}|}{\Delta_i}, M = \begin{cases} r\mu, \mu > 0 \\ 1, \mu = 0 \end{cases} \tag{3.1}$$

где r является заданным параметром метода, а $\Delta_i = (x_i - x_{i-1})^{\frac{1}{N}}$.

Шаг 3. Для каждого интервала $(x_{i-1}, x_i), 1 \leqslant i \leqslant k+1$, вычислить характеристику в соответствии с формулами

$$R(1) = 2\Delta_1 - 4\frac{z_1}{M}, R(k+1) = 2\Delta_{k+1} - 4\frac{z_k}{M} \tag{3.2}$$

$$R(i) = \Delta_i + \frac{(z_i - z_{i-1})^2}{M^2 \Delta_i} - 2\frac{z_i + z_{i-1}}{M}, 1 < i < k+1 \tag{3.3}$$

Шаг 4. Выбрать наибольшую характеристику:

$$t = \arg\max_{1 \leqslant i \leqslant k+1} R(i) \tag{3.4}$$

Шаг 5. Провести очередное испытания в точке x_{k+1}, вычисленной по формулам

$$x_{k+1} = \frac{x_t + x_{t-1}}{2}, t = 1, t = k+1$$

$$x_{k+1} = \frac{x_t + x_{t-1}}{2} - \mathrm{sign}(z_t - z_{t-1})\frac{1}{2r}\left[\frac{|z_t - z_{t-1}|}{\mu}\right]^N, 1 < t < k+1 \tag{3.5}$$

Алгоритм прекращает работу, если выполняется условие $\Delta_t \leqslant \varepsilon$, где $\varepsilon > 0$ есть заданная точность. В качестве оценки глобально-оптимального решения задачи выбираются значения

$$f_k^* = \min_{1 \leqslant i \leqslant k} f(x_i), x_k^* = \arg\min_{1 \leqslant i \leqslant k} f(x_i) \tag{3.6}$$

Подробнее метод и теорема о его сходимости описаны в [4].

3.1. Сравнение методов оптимизации

Существует несколько критериев оптимальности алгоритмов поиска (минимаксный, критерий одношаговой оптимальности), но большинстве случаев представляет интерес сравнение методов по среднему результату, достижимому на конкретном подклассе липшицевых функций. Достоинством такого подхода является то, что средний показатель можно оценить по конечной случайной выборке задач, используя методы математической статистики.

В качестве оценки эффективности алгоритма будем использовать, операционную характеристику, которая определяется множеством точек на плоскости (K, P), где K – среднее число поисковых испытаний, предшествующих выполнению условия остановки при минимизации функции из данного класса, а P – статистическая вероятность того, что к моменту остановки глобальный экстремум будет найден с заданной точностью. Если при выбранном K операционная характеристика одного метода лежит выше характеристики другого, то это значит, что при фиксированных затратах на поиск первый метод найдёт решение с большей статистической вероятностью. Если же зафиксировать некоторое значение P, и характеристика одного метода лежит левее характеристики другого, то первый метод требует меньше затрат на достижение той же надёжности.

3.2. Класс тестовых задач GKLS

Для сравнения алгоритмов глобального поиска в смысле операционной характеристики требуется иметь некоторое множество тестовых задач. Генератор задач GKLS, описанный в [5] позволяет получить такое множество задач с заренее известными свойствами. В данной работе используются два класса, сгенерированные GKLS: 4d Simple и 5d Simple, параметры которых также описаны в [5]. Функции рассматриваемых классов являются непрерывно дифференцируемыми и имеют 10 локальных минимумов, один из которых является глобальным.

4. Многоуровневая схема редукции размерности с помощью разверток

Теоретически с помощью развёрток можно решить задучю любой размерности, однако на ЭВМ развёртка строится с помощью конечноразрядной арифметики, из-за чего, начиная с некоторого N^*, построение разветки невозможно (значение N^* зависит от максимального количества значащих разрядов в арифметике с плавающей точкой). Понять почему это происходит нетрудно, обратившись, например к [4].

Чтобы преодолеть эту проблему профессором В. П. Гергелем была предложена следующая идея: использовать композицию развёрток меньшей размерности для построения отображения $z(x) : [0; 1] \rightarrow D \in R^N$. Поясним эту схему на примере редукции размерности в четырёхмерной задаче. Пусть $y_2(x)$ — двухмерная развёртка (отображает отрезок в прямоугольник), тогда рассмотрим функцию $\psi(x_1, x_2) = \phi(y_2(x_1), y_2(x_2))$. К $\psi(x_1, x_2)$ можно также применить редукцию размерности с помощью развёртки. Таким образом, задав точку $x^* \in [0; 1]$, вычислив $y_2(x^*) = (x_1, x_2)$ и пару векторов $(y_2(x_1), y_2(x_2))$, получим четырёхмерную точку. Из инъективности $y_2(x)$ следует инъективность $z(x)$.

Проблемой этого подхода является выяснение свойств функции $\phi(z(x))$ и возможности использования одномерного метода Стронгина с гёльдеровой метрикой для оптимизации $\phi(z(x))$. Чтобы не тратить время на теоретическое исследование, были проведены численные эксперименты с целью оценить возможность применения многоуровневой развёртки в четырёхмерном случае.

Прежде всего, рассмотрим линии уровня функции $\psi(x_1, x_2) = \phi(y_2(x_1), y_2(x_2))$ при $\phi(t) = \sum_{i=1}^{4}(t_i - 0.5)^2$. Как видно из рис. 4.1, линии уровня имеют довольно сложную струк-

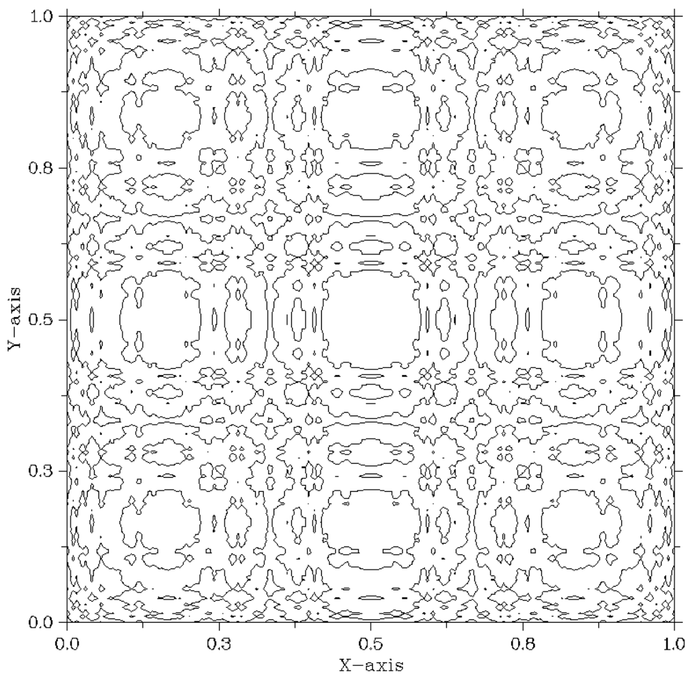

Рис. 4.1: Линии уровня функции $\psi(x_1, x_2) = \phi(y_2(x_1), y_2(x_2))$

туру, что говорит о возможных сложностях применения одномерного метода с разверткой.

Далее был проведён более масштабный вычислительный экспеимент: с помощью многоуровневой развертки решались 100 задач из класса GKLS 4d Simple. На рис. 4.2 приведены операционные характеристики метода с простой и многоуровневой развёртками. При этом были зафиксированы следующие параметры алгоритма: надёжность $r = 4.5$, плотность построения всех развёрток 12, критерий остановки попадание точки, поставленной методом в квадрат со стороной $\varepsilon = 10^{-2}$, центром которого является решение задачи. Как видно из

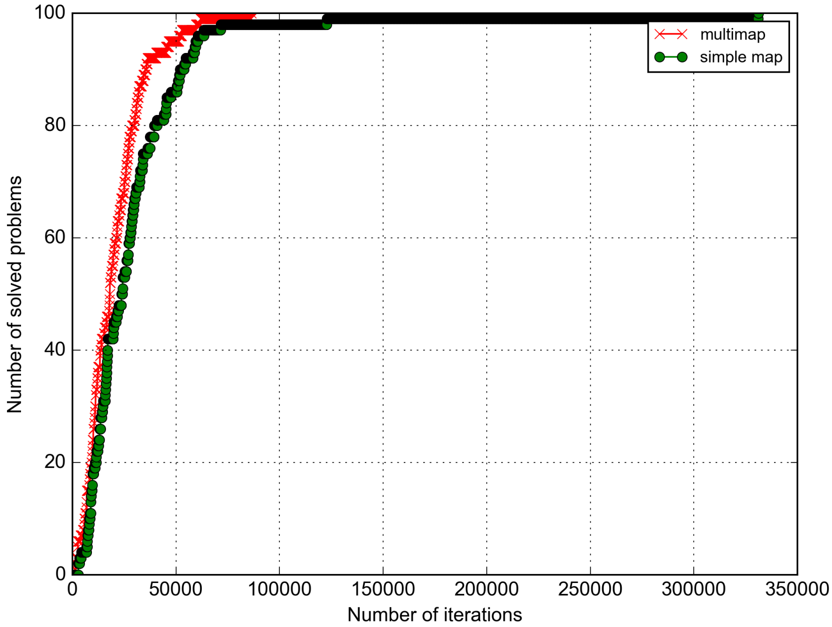

Рис. 4.2: Операционная характеристика метода с различными развёртками на классе GKLS 4d Simple

графика, операционная характеристика метода с многоуровневой развёрткой лежит выше, чем аналогичная кривая метода с простой развёрткой. Кроме того, метод с многоуровневой развёрткой заметно раньше вышел на стопроцентную надёжность на решаемой выборке задач. Исходя из первых экспериментов можно сказать, что при данных параметрах алгоритма многоуровневая развертка лучше, но при уменьшении точности алгоритма ε до значения 10^{-3} выполнения критерия остановки в случае использования многоуровневой развёртки с выбранной плотностью построения внутренних развёрток не происходит. Если плотность увеличить до 16, то остановка также не будет происходить. Исходя из этого можно делать вывод, что использование на практике многоуровневых развёрток с большой долей вероятности невозможно из-за описанного дефекта сходимости.

5. Применение локального поиска для ускорения сходимости АГП

Методы локального поиска могут применяться в сочетании с глобальными алгоритмами для улучшения полученных решений или текущих оценок оптимума. В первом случае локальный метод стартует из точки, найденной глобальным методом, и уточняет решение практически до любой нужной точности. Это позволяет избежать чрезмерных затрат на поиск решения с высокой точностью глобальным методом.

Во втором случае локальный метод используется для ускорения обнаружения локальных оптимумов. Информацилнно-статистический метод Стронгина позволяет обновлять свою поисковую информацию из любых посторонних источников, в том числе из точкек испытаний, полученных от локального метода. Как только глобальный метод находит новую оценку оптимума, из этой точки стартует локальный метод и все или часть испытаний, проведённых им добавляется в поисковую информацию, далее глобальный метод продолжает работу. Каких-либо теоретических исследований подобной схемы не проводилось, поэтому её эффективность проверялась экспериментально.

В качестве метода локальной оптимизации был выбран метод Хука-Дживса [6]. Он прост в реализации и для его работы не требуется знать значений производных оптимизируемой функции.

Были проведены две серии экспериментов, соответствующих следующим схемам добавления точек, полученных локальным методом в поисковую информацию:

- добавление единственной точки, к которой сошёлся локальный метод;
- добавление всех промежуточных точек.

Эксперименты проводились на классах GKLS 4d Simple и GKLS 5d Simple, параметры метода были заданы такие же, как в разделе 4. Из рис. 5.3, 5.4 можно сделать вывод, что ва-

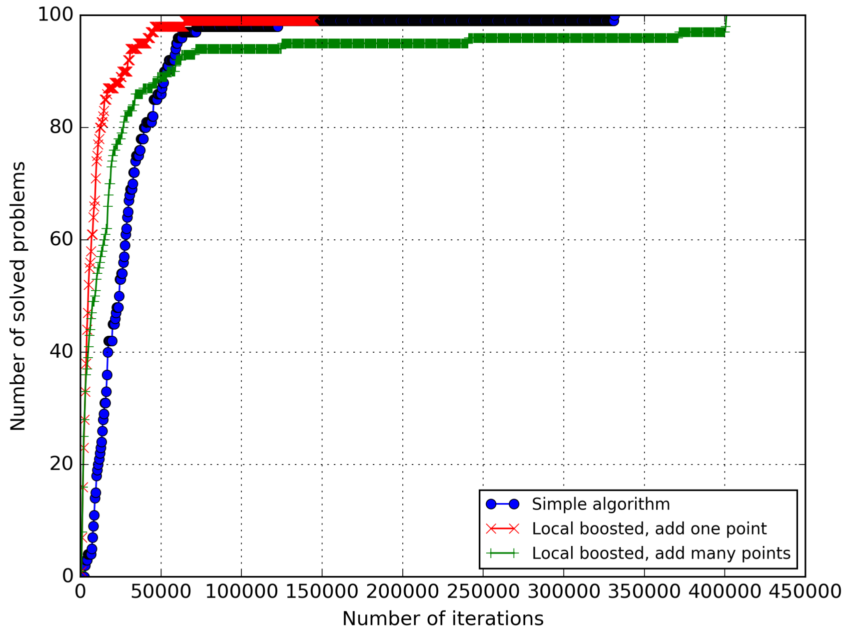

Рис. 5.3: Операционные характеристики на классе GKLS 4d Simple при различных вариантах использования локального метода

риант с использованием только одной лучшей точки, полученной локальным методом, оказался наилучшим. В обоих случаях он заметно ускорил сходимость глобального алгоритма.

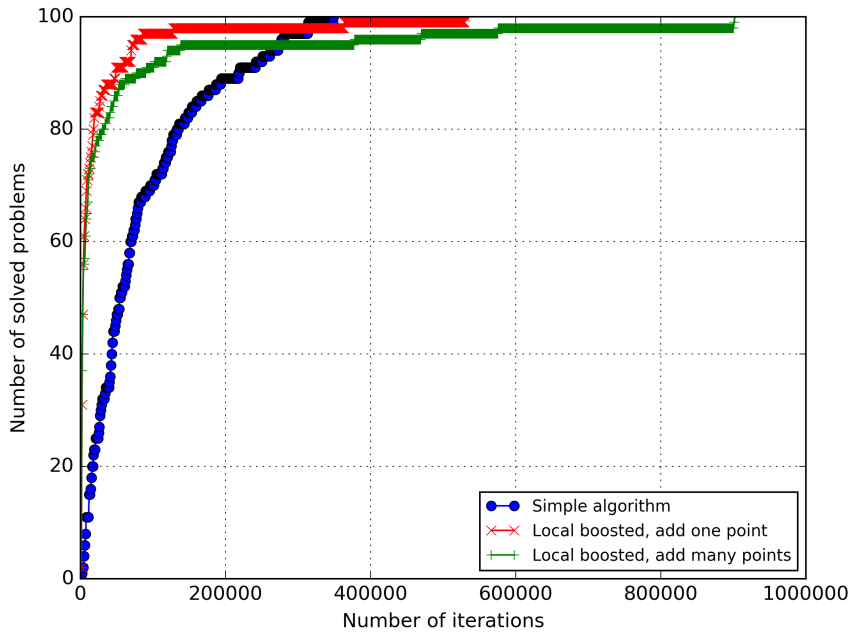

Рис. 5.4: Операционные характеристики на классе GKLS 5d Simple при различных вариантах использования локального метода

6. Смешанный алгоритм глобального поиска и его эффективная реализация

Ещё одной модификацией метода Стронгина, позволяющей в процессе оптимизации лучше учитывать данные о локальных оптимумах, найденных в процессе поиска, является смешанный алгоритм Стронгина-Маркина [7]. Наряду с характеристикой интервала $R(i)$ (3.3) можно рассматривать $R^*(i)$, которая будет более чувствиетльна к наличию в интревале текущего найденного минимума функции x^*_k:

$$R^*(i) = \frac{R(i)}{\sqrt{(z_i - z^*)(z_{i-1} - z^*)}/\mu + 1.5^{-\alpha}}$$

где $f(x^*_k) = z^*$, а $\alpha \in [1; 30]$ — степень локальности. Чем она больше, тем более высокая характеристика у интервала, содержащего x^*_k, по сравнению с остальными.

Смешанный алгоритм состоит в следующем: в процессе работы метода каждые S итераций интервал для последующего разбиения выбирается по характеристикам $R^*(i)$. S — параметр смешивания. Такой подход позволяет существенно ускорить сходимость метода. На рис. 6.5 приведены операционные характеристики чисто глобального и смешанного алгоритма на классе GKLS 4d Simple. Параметр смешивания S равен 5, $\alpha = 15$, остальные параметры метода были заданы такие же, как в разделе 4.

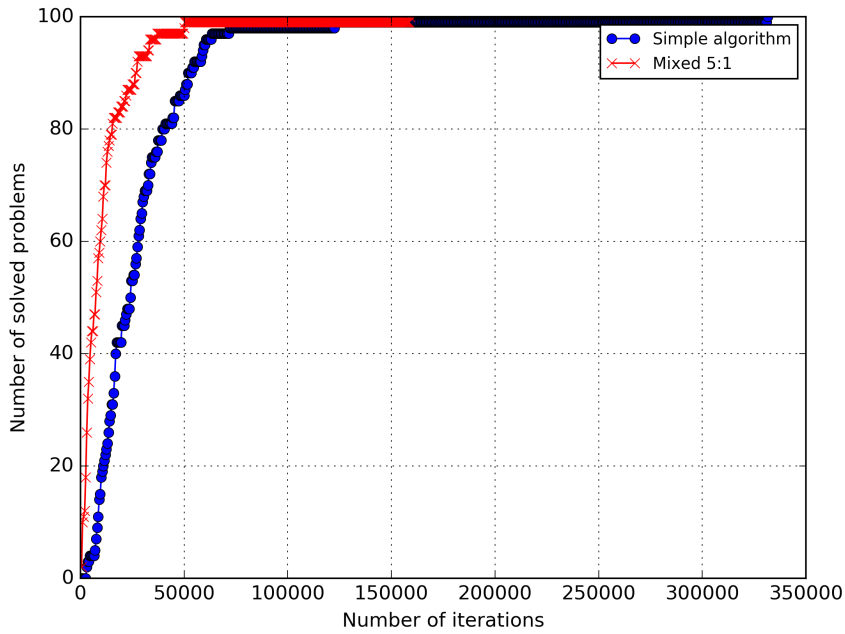

Рис. 6.5: Операционные характеристики обычного и смешанного АГП на классе GKLS 4d Simple

Из-за того, что интервал имеет сразу две характеристики, появляется проблема эффективной реализации смешанного алгоритма. Если интервал имеет одну характеристику, то для выбора максимальной достаточно организовать приоритетную очередь характеристик [8]. Причём перезаполнение такой очереди необходимо не на каждой итерации: в большинстве случаев характеристики интервалов не меняются, достаточно удалить разбиваемый интервал и вставить в очередь два новых. Такая организация работы метода позволяет существенно сократить объём вычислений. При наличии у интервала двух характеристик можно организовать две связанные очереди. В этом случае необходимо предусмотреть процедуру синхронизации двух очередей.

Перечислим операции, при которых необходима синхронизаия:

- вставка интервала сразу в обе очереди;
- удаление интервала из какой-либо очереди.

Синхронизация достигается путём введения перекрёстных ссылок между элементами очередей. На рис. 6.6 приведена схема связанных очередей. Элемент очереди представляет собой совокупность ключа (**LocalR** или **R**), указателя на интервал (**pInterval**) и указателя на элемент связанной очереди, соответствующий тому же интервалу (**pLinkedElement**). Опишем подробнее алгоритмы вставки и удаления элементов.

Вставка элемента в пару связанных очередей:

1. Попытаться вставить элемент в очередь глобальных характеристик (он может быть не

вставлен, если имеет слишком низкий приоритет).

2. Попытаться вставить элемент в очередь локальных характеристик (он может быть не вставлен, если имеет слишком низкий приоритет).

3. Если элемент интервал вставлен в обе очереди, то выставить перекрёстные ссылки.

Удаление элемента с минимальным ключом:

1. Удалить элемент с минимальным ключом из очереди, запомнить указатель **pLinkedElement**.

2. Если **pLinkedElement** ненулевой, то вызвать процедуру удаления элемента, на который указывает **pLinkedElement** в структуре данных, хранящей связянную очередь.

Последний момент, который надо учесть при реализации: вставка или удаление элемента очереди приводит к тому, что необходимо восстановить её внутреннюю структуру. Если очередь хранится в куче, то восстанавливается свойство кучеобразности. Во время этого процесса требуется производить попарные перестановки элементов, а значит, необходимо обновлять ссылки на эти элементы в связанной очереди.

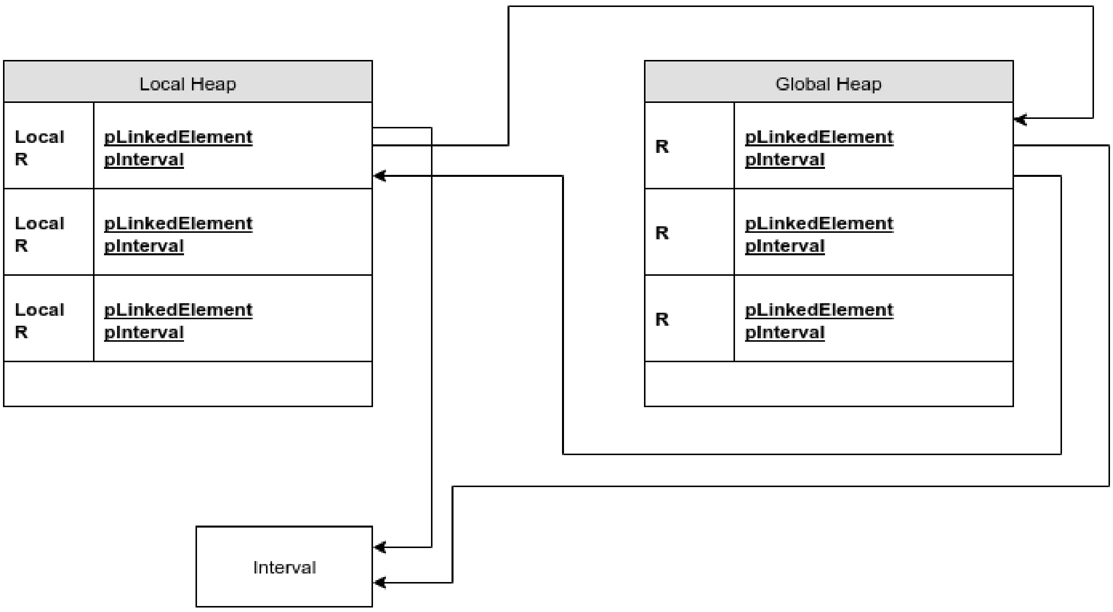

Рис. 6.6: Схема устройства связянных очередей

Стоит заметить, что внесённые модификации (в основном эта работа со ссылками) не увеличивают ассимптотическую сложность выполнения операций вставки и удаления по сравнению с единственной очередью.

7. Заключение

В ходе работы были получены следующие практические результаты:

- реализован метод локальной оптимизации Хука-Дживса (код в приложении 8.1.);
- в системе ExaMin реализованы различные стратегии использования локального поиска;
- в рамках ExaMin реализована поддержка смешанного алгоритма глобального поиска, а также эффективные структуры данных, необходимые для этого алгоритма (фрагменты кода в приложении 8.2.)
- был проведён отдельный эксперимент для выяснения возможности практического использования многоуровневых развёрток.

Список литературы

1. *А.В. Сысоев, К.А. Баркалов, В.П. Гергель* Блочная многошаговая схема параллельного решения задач многомерной глобальной оптимизации // Материалы XIV Международной конференции ”Высокопроизводительные параллельные вычисления на кластерных системах 10-12 ноября, ПНИПУ, Пермь. — 2014. — С. 425—432.
2. *Сысоев А.В. Баркалов К.А. Гергель В.П. Лебедев И.Г.* MPI-реализация блочной многошаговой схемы параллельного решения задач глобальной оптимизации // Суперкомпьютерные дни в России: Труды международной конференции (28-29 сентября 2015 г., г. Москва). — М.: Изд-во МГУ, 2015. — С. 411—419.
3. *К.А. Баркалов, И.Г. Лебедев, В.В. Соврасов, А.В. Сысоев* Реализация параллельного алгоритма поиска глобального экстремума функции на Intel Xeon Phi // Вычислительные методы и программирование. — 2016. — Т. 17. — С. 101—110.
4. *Стронгин Р. Г.* «Численные методы в многоэкстремальных задачах (информационно-статистические алгоритмы)». — «Наука», М., 1978, 240 стр.
5. *Квасов Д.Е., Сергеев Я.Д.* Исследование методов глобальной оптимизации при помощи генератора классов тестовых функций. — Н.Новгород: Изд-во ННГУ, 2011.
6. *Химмельблау Д. М.* Прикладное нелинейное программирование. — Издательство МИР, Москва, 1975.
7. *Д. Л. Маркин, Р. Г. Стронгин* Метод решения многоэкстремальных задач с невыпуклыми ограничениями, использующий априорную информацию об оценках оптимума // Ж. вычисл. матем. и матем. физ. — 1987. — Т. 27:1. — С. 52—62.
8. *M. D. Atkinson, J.R. Sack, N. Santoro, T. Strothotte* Min-Max Heaps and Generalized Priority Queues // Communications of the ACM. — 1986. — Т. 29. — С. 996—1000.

8. Приложения

8.1. Приложение 1

```
#ifndef __LOCALMETHOD_H__
#define __LOCALMETHOD_H__

#include "parameters.h"
#include "task.h"
#include "data.h"
#include "common.h"
#include <vector>

#define MAX_LOCAL_TRIALS_NUMBER 10000

class TLocalMethod
{
protected:

  int mDimension;
  int mConstraintsNumber;
  int mTrialsCounter;
  int mMaxTrial;

  TTrial mBestPoint;
  std::vector<TTrial> mSearchSequence;
  TTask* mPTask;

  bool mIsLogPoints;

  double mEps;
  double mStep;
  double mStepMultiplier;

  OBJECTIV_TYPE *mFunctionsArgument;
  OBJECTIV_TYPE *mStartPoint;
  OBJECTIV_TYPE* mCurrentPoint;
  OBJECTIV_TYPE* mCurrentResearchDirection;
  OBJECTIV_TYPE* mPreviousResearchDirection;

  double MakeResearch(OBJECTIV_TYPE*);
  void DoStep();
  double EvaluateObjectiveFunctiuon(const OBJECTIV_TYPE*);

public:

  TLocalMethod();
  TLocalMethod(TParameters _params, TTask* _pTask, TTrial _startPoint, bool
   logPoints = false);
  ~TLocalMethod();

  void SetEps(double);
  void SetInitialStep(double);
  void SetStepMultiplier(double);
  void SetMaxTrials(int);

  int GetTrialsCounter() const;
  std::vector<TTrial> GetSearchSequence() const;

  TTrial StartOptimization();
```

```
};

#endif //__LOCALMETHOD_H__
```

```
#include ”local_method.h”
#include <cmath>
#include <cstring>
#include <algorithm>

TLocalMethod::TLocalMethod(TParameters _params, TTask* _pTask, TTrial
    _startPoint, bool logPoints)
{
  mEps = _params.Epsilon / 100;
  if (mEps > 0.0001)
    mEps = 0.0001;
  mBestPoint = _startPoint;
  mStep = _params.Epsilon * 2;
  mStepMultiplier = 2;
  mTrialsCounter = 0;
  mIsLogPoints = logPoints;

  mPTask = _pTask;

  mDimension = mPTask->GetN() - mPTask->GetFixedN();
  mConstraintsNumber = mPTask->GetNumOfFunc() - 1;

  mStartPoint = new OBJECTIV_TYPE[mDimension];
  std::memcpy(mStartPoint,
    _startPoint.y + mPTask->GetFixedN(),
    mDimension * sizeof(OBJECTIV_TYPE));

  mFunctionsArgument = new OBJECTIV_TYPE[mPTask->GetN()];
  std::memcpy(mFunctionsArgument,
    _startPoint.y,
    mPTask->GetN() * sizeof(OBJECTIV_TYPE));
  mMaxTrial = MAX_LOCAL_TRIALS_NUMBER;
}

TLocalMethod::TLocalMethod() : mPTask(NULL), mStartPoint(NULL),
mFunctionsArgument(NULL), mMaxTrial(MAX_LOCAL_TRIALS_NUMBER)
{
}

TLocalMethod::~TLocalMethod()
{
  if (mStartPoint)
    delete[] mStartPoint;
  if (mFunctionsArgument)
    delete[] mFunctionsArgument;
}

void TLocalMethod::DoStep()
{
  for (int i = 0; i < mDimension; i++)
    mCurrentPoint[i] = (1 + mStepMultiplier)*mCurrentResearchDirection[i] -
    mStepMultiplier*mPreviousResearchDirection[i];
}

TTrial TLocalMethod::StartOptimization()
```

```
{
  int k = 0, i = 0;
  bool needRestart = true;
  double currentFValue, nextFValue;
  mTrialsCounter = 0;

  mCurrentPoint = new OBJECTIV_TYPE[mDimension];
  mCurrentResearchDirection = new OBJECTIV_TYPE[mDimension];
  mPreviousResearchDirection = new OBJECTIV_TYPE[mDimension];

  while (mTrialsCounter < mMaxTrial) {
    i++;
    if (needRestart) {
      k = 0;
      std::memcpy(mCurrentPoint, mStartPoint, sizeof(OBJECTIV_TYPE)*mDimension);
      std::memcpy(mCurrentResearchDirection, mStartPoint, sizeof(OBJECTIV_TYPE)*mDimension);
      currentFValue = EvaluateObjectiveFunctiuon(mCurrentPoint);
      needRestart = false;
    }

    std::swap(mPreviousResearchDirection, mCurrentResearchDirection);
    std::memcpy(mCurrentResearchDirection, mCurrentPoint, sizeof(OBJECTIV_TYPE)*mDimension);
    nextFValue = MakeResearch(mCurrentResearchDirection);

    if (currentFValue > nextFValue) {
      DoStep();

      if (mIsLogPoints)
      {
        TTrial currentTrial;
        currentTrial.index = mBestPoint.index;
        currentTrial.FuncValues[currentTrial.index] = nextFValue;
        std::memcpy(currentTrial.y, mFunctionsArgument, sizeof(OBJECTIV_TYPE)*mPTask->GetFixedN());
        std::memcpy(currentTrial.y + mPTask->GetFixedN(), mCurrentPoint, sizeof(OBJECTIV_TYPE)*mDimension);
        mSearchSequence.push_back(currentTrial);
      }
      k++;
      currentFValue = nextFValue;
    }
    else if (mStep > mEps) {
      if (k != 0)
        std::memcpy(mStartPoint, mPreviousResearchDirection, sizeof(OBJECTIV_TYPE)*mDimension);
      else
        mStep /= mStepMultiplier;
      needRestart = true;
    }
    else
      break;
  }

  if (currentFValue < mBestPoint.FuncValues[mConstraintsNumber])
  {
    std::memcpy(mBestPoint.y + mPTask->GetFixedN(),
      mPreviousResearchDirection, sizeof(OBJECTIV_TYPE)*mDimension);
    mBestPoint.FuncValues[mConstraintsNumber] = currentFValue;
```

```
    mSearchSequence.push_back(mBestPoint);
  }

  delete[] mCurrentPoint;
  delete[] mPreviousResearchDirection;
  delete[] mCurrentResearchDirection;

  return mBestPoint;
}

double TLocalMethod::EvaluateObjectiveFunctiuon(const OBJECTIV_TYPE* x)
{
  if (mTrialsCounter >= mMaxTrial)
    return HUGE_VAL;

  for (int i = 0; i < mDimension; i++)
    if (x[i] < mPTask->GetA()[mPTask->GetFixedN() + i] ||
      x[i] > mPTask->GetB()[mPTask->GetFixedN() + i])
      return HUGE_VAL;

  std::memcpy(mFunctionsArgument + mPTask->GetFixedN(), x, mDimension * sizeof
   (OBJECTIV_TYPE));
  for (int i = 0; i <= mConstraintsNumber; i++)
  {
    double value = mPTask->GetFuncs()[i](mFunctionsArgument);
    if (i < mConstraintsNumber && value > 0)
    {
      mTrialsCounter++;
      return HUGE_VAL;
    }
    else if (i == mConstraintsNumber)
    {
      mTrialsCounter++;
      return value;
    }
  }

  return HUGE_VAL;
}

void TLocalMethod::SetEps(double eps)
{
  mEps = eps;
}

void TLocalMethod::SetInitialStep(double value)
{
  mStep = value;
}

void TLocalMethod::SetStepMultiplier(double value)
{
  mStepMultiplier = value;
}

void TLocalMethod::SetMaxTrials(int count)
{
  mMaxTrial = std::min(count, MAX_LOCAL_TRIALS_NUMBER);
}

int TLocalMethod::GetTrialsCounter() const
```

```
{
  return mTrialsCounter;
}

std::vector<TTrial> TLocalMethod::GetSearchSequence() const
{
  return mSearchSequence;
}

double TLocalMethod::MakeResearch(OBJECTIV_TYPE* startPoint)
{
  double bestValue = EvaluateObjectiveFunctiuon(startPoint);

  for (int i = 0; i < mDimension; i++)
  {
    startPoint[i] += mStep;
    double rightFvalue = EvaluateObjectiveFunctiuon(startPoint);

    if (rightFvalue > bestValue)
    {
      startPoint[i] -= 2 * mStep;
      double leftFValue = EvaluateObjectiveFunctiuon(startPoint);
      if (leftFValue > bestValue)
        startPoint[i] += mStep;
      else
        bestValue = leftFValue;
    }
    else
      bestValue = rightFvalue;
  }

  return bestValue;
}
```

8.2. Приложение 2

```
#ifndef __DUAL_QUEUE_H__
#define __DUAL_QUEUE_H__

#include ”minmaxheap.h”
#include ”common.h”
#include ”queue_common.h”

class TPriorityDualQueue : public TPriorityQueueCommon
{
protected:
  int MaxSize;
  int CurLocalSize;
  int CurGlobalSize;

  MinMaxHeap< TQueueElement, _less >* pGlobalHeap;
  MinMaxHeap< TQueueElement, _less >* pLocalHeap;

  void DeleteMinLocalElem();
  void DeleteMinGlobalElem();
  void ClearLocal();
  void ClearGlobal();
public:
```

```
  TPriorityDualQueue(int _MaxSize = DefaultQueueSize); // _MaxSize must be
    qual to 2^k - 1
  ~TPriorityDualQueue();

  int GetLocalSize() const;
  int GetSize() const;
  int GetMaxSize() const;
  bool IsLocalEmpty() const;
  bool IsLocalFull() const;
  bool IsEmpty() const;
  bool IsFull() const;

  void Push(double globalKey, double localKey, void *value);
  void PushWithPriority(double globalKey, double localKey, void *value);
  void Pop(double *key, void **value);

  void PopFromLocal(double *key, void **value);

  void Clear();
  void Resize(int size);
};
#endif
```

```
#include "dual_queue.h"

TPriorityDualQueue::TPriorityDualQueue(int _MaxSize)
{
  MaxSize = _MaxSize;
  CurGlobalSize = CurLocalSize = 0;
  pLocalHeap = new MinMaxHeap< TQueueElement, _less >(MaxSize);
  pGlobalHeap = new MinMaxHeap< TQueueElement, _less >(MaxSize);
}

TPriorityDualQueue::~TPriorityDualQueue()
{
  delete pLocalHeap;
  delete pGlobalHeap;
}

void TPriorityDualQueue::DeleteMinLocalElem()
{
  TQueueElement tmp = pLocalHeap->popMin();
  CurLocalSize--;

  //update linked element in the global queue
  if (tmp.pLinkedElement != NULL)
    tmp.pLinkedElement->pLinkedElement = NULL;
}

void TPriorityDualQueue::DeleteMinGlobalElem()
{
  TQueueElement tmp = pGlobalHeap->popMin();
  CurGlobalSize--;

  //update linked element in the local queue
  if (tmp.pLinkedElement != NULL)
    tmp.pLinkedElement->pLinkedElement = NULL;
}
```

```
int TPriorityDualQueue::GetLocalSize() const
{
  return CurLocalSize;
}

int TPriorityDualQueue::GetSize() const
{
  return CurGlobalSize;
}

int TPriorityDualQueue::GetMaxSize() const
{
  return MaxSize;
}

bool TPriorityDualQueue::IsLocalEmpty() const
{
  return CurLocalSize == 0;
}

bool TPriorityDualQueue::IsLocalFull() const
{
  return CurLocalSize == MaxSize;
}

bool TPriorityDualQueue::IsEmpty() const
{
  return CurGlobalSize == 0;
}

bool TPriorityDualQueue::IsFull() const
{
  return CurGlobalSize == MaxSize;
}

void TPriorityDualQueue::Push(double globalKey, double localKey, void * value)
{
  TQueueElement* pGlobalElem = NULL, *pLocalElem = NULL;
  //push to a global queue
  if (!IsFull()) {
    CurGlobalSize++;
    pGlobalElem = pGlobalHeap->push(TQueueElement(globalKey, value));
  }
  else {
    if (globalKey > pGlobalHeap->findMin().Key) {
      DeleteMinGlobalElem();
      CurGlobalSize++;
      pGlobalElem = pGlobalHeap->push(TQueueElement(globalKey, value));
    }
  }
  //push to a local queue
  if (!IsLocalFull()) {
    CurLocalSize++;
    pLocalElem = pLocalHeap->push(TQueueElement(localKey, value));
  }
  else {
    if (localKey > pLocalHeap->findMin().Key) {
      DeleteMinLocalElem();
      CurLocalSize++;
      pLocalElem = pLocalHeap->push(TQueueElement(localKey, value));
```

```
    }
  }
  //link elements
  if (pGlobalElem != NULL && pLocalElem != NULL) {
    pGlobalElem->pLinkedElement = pLocalElem;
    pLocalElem->pLinkedElement = pGlobalElem;
  }
}

void TPriorityDualQueue::PushWithPriority(double globalKey, double localKey,
    void * value)
{
  TQueueElement* pGlobalElem = NULL, *pLocalElem = NULL;
  //push to a global queue
  if (!IsEmpty()) {
    if (globalKey > pGlobalHeap->findMin().Key) {
      if (IsFull())
        DeleteMinGlobalElem();
      CurGlobalSize++;
      pGlobalElem = pGlobalHeap->push(TQueueElement(globalKey, value));
    }
  }
  else {
    CurGlobalSize++;
    pGlobalElem = pGlobalHeap->push(TQueueElement(globalKey, value));
  }
  //push to a local queue
  if (!IsLocalEmpty()) {
    if (localKey > pLocalHeap->findMin().Key) {
      if (IsLocalFull())
        DeleteMinLocalElem();
      CurLocalSize++;
      pLocalElem = pLocalHeap->push(TQueueElement(localKey, value));
    }
  }
  else {
    CurLocalSize++;
    pLocalElem = pLocalHeap->push(TQueueElement(localKey, value));
  }
  //link elements
  if (pGlobalElem != NULL && pLocalElem != NULL) {
    pGlobalElem->pLinkedElement = pLocalElem;
    pLocalElem->pLinkedElement = pGlobalElem;
  }
}

void TPriorityDualQueue::PopFromLocal(double * key, void ** value)
{
  TQueueElement tmp = pLocalHeap->popMax();
  *key = tmp.Key;
  *value = tmp.pValue;
  CurLocalSize--;

  //delete linked element from the global queue
  if (tmp.pLinkedElement != NULL) {
    pGlobalHeap->deleteElement(tmp.pLinkedElement);
    CurGlobalSize--;
  }
}

void TPriorityDualQueue::Pop(double * key, void ** value)
```

```
{
  TQueueElement tmp = pGlobalHeap->popMax();
  *key = tmp.Key;
  *value = tmp.pValue;
  CurGlobalSize--;

  //delete linked element from the local queue
  if (tmp.pLinkedElement != NULL) {
    pLocalHeap->deleteElement(tmp.pLinkedElement);
    CurLocalSize--;
  }
}

void TPriorityDualQueue::Clear()
{
  ClearLocal();
  ClearGlobal();
}

void TPriorityDualQueue::Resize(int size)
{
  MaxSize = size;
  CurGlobalSize = CurLocalSize = 0;
  delete pLocalHeap;
  delete pGlobalHeap;
  pLocalHeap = new MinMaxHeap< TQueueElement, _less >(MaxSize);
  pGlobalHeap = new MinMaxHeap< TQueueElement, _less >(MaxSize);
}

void TPriorityDualQueue::ClearLocal()
{
  pLocalHeap->clear();
  CurLocalSize = 0;
}

void TPriorityDualQueue::ClearGlobal()
{
  pGlobalHeap->clear();
  CurGlobalSize = 0;
}
```