МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«АЛТАЙСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
Факультет математики и информационных технологий
Кафедра информатики

РАСПРЕДЕЛЕННЫЙ РЕШАТЕЛЬ ЗАДАЧ ГЛОБАЛЬНОЙ ОПТИМИЗАЦИИ

(магистерская диссертация)

Выполнил:
студент 447М-ВМ группы
Косолапов Л.Ю.
_____

Научный руководитель:
к.ф.-м.н., доцент,
Жилин С.И.
_____

Работа защищена:

«___» _____ 2016 г.

Оценка: _____

Председатель ГЭК:

_____

_____

Ф.И.О

Допустить к защите:
ВРИО зав. каф., к.ф.-м.н.,
Жариков А.В.

_____

«___» _____ 2016 г.

Барнаул – 2016

Реферат

Тема магистерской работы: «Распределенный решатель задач глобальной оптимизации».

Цель работы − создание распределенного решателя задач глобальной оптимизации функции многих переменных.

Объект работы − параллельные алгоритмы глобальной оптимизации.

Предмет работы – способы распределенной реализации алгоритмов глобальной оптимизации.

В рамках работы были рассмотрены алгоритмы глобальной оптимизации, проведен обзор программных инструментов для глобальной оптимизации и систем построения распределенных вычислений с целью их пригодности для реализации распределенного решателя задач глобальной оптимизации. На основе выбранных инструментов был разработан распределенный решатель задач глобальной оптимизации и проведены вычислительные эксперименты для тестирования вычислительной корректности и выяснения условий эффективного использования решателя задач глобальной оптимизации.

Результатом работы является реализованное приложение «Распределенный решатель задач глобальной оптимизации».

Объем работы 81 страница, количество рисунков – 26, листингов – 18, таблиц – 3, приложений – 1, использованных источников литературы − 34.

Ключевые слова: глобальная оптимизация, распределенные вычисления, интервальные алгоритмы глобальной оптимизации.

Содержание

Введение

Решение задач оптимизации – одно из самых перспективных, но в тоже время трудоемких направлений современной вычислительной и прикладной математики. Решение таких проблем требуется во многих задачах из различных отраслей науки и техники.

Рассматривается задача глобальной оптимизации вещественнозначной функции $f : X \to \mathbb{R}$ на прямоугольном брусе $X \subset \mathbb{R}^n$ со сторонами, параллельными координатным осям. Необходимо найти:

$$\min_{x \in X} f(x).$$

Частным случаем задач оптимизации являются задачи поиска глобальных экстремумов функции, т.е. таких, которые являются самыми лучшими на всей рассматриваемой области определения заданной функции, а не только в сравнении с близлежащими точками из некоторой своей окрестности. Такие задачи представляют особый интерес. Следует отметить, что задача поиска глобального оптимума, в общем случае, является труднорешаемой. Классические численные методы, как правило, сводятся к поиску всех локальных экстремумов функции и затем сравнению их между собой. Например, для полиномиальной целевой функции $f(x)$ задача глобальной оптимизации на прямоугольном брусе является *NP*-трудной, т.е. фактически для решения требуются экспоненциальные в зависимости от размерности трудозатраты. Это было строго показано А.А. Гагановым в 1984 году [1]. Известна также теорема Кирфотта-Крейновича [2], утверждающая, что за пределами класса выпуклых целевых функций поиск глобального экстремума является *NP*-трудным.

Многие задачи, возникающие в различных сферах человеческой деятельности, могут быть сведены к задаче поиска глобального оптимума. Оптимизация − это важная составляющая этапов моделирования экономических, технических, экологических и других систем.

В настоящее время глобальная оптимизация – широко востребованное и интенсивно развивающееся направление вычислительной математики.

Трудности поиска глобального экстремума во многом связаны с видом оптимизируемой функции и количеством ее аргументов. Целевая функция может быть недифференцируемой, невыпуклой, негладкой, многоэкстремальной. Таким образом для каждого вычисления значения целевой функции может потребоваться большое количество вычислительных ресурсов. Часто на практике требуется не просто приближённое решение, но и гарантия его максимальной близости к идеальному экстремуму на всей рассматриваемой области определения. Такая постановка задачи называются «доказательная глобальная оптимизация», и она является чрезвычайно трудной.

На данный момент имеется богатый набор инструментов и готового программного обеспечения реализованного для поиска глобального экстремума функции, каждое из которых предназначено для определенного класса задач [3,4]. Одни решают задачи только на целевой функции малой размерности, другие только на выпуклых функциях, третьи предназначены для нелинейной оптимизации на непрерывных функциях и т.д. Каждое программное обеспечение имеет свои достоинства и недостатки, но в связи с возрастающей потребностью в вычислительных ресурсах, при решении задач глобальной оптимизации, возникает проблема эффективности использования программного обеспечения работающего в пределах системы с общей памятью (фактически одной машины, возможно с многоядерным процессором).

Часто возникает задача, решение которой требует значительного времени и значительных ресурсов, обеспечение которых на одном вычислительном узле неоправданно дорого. В связи с этим выявляется потребность в реализации программных средств, которые позволяют масштабировать решение задачи глобальной оптимизации на распределенной системе, то есть совокупности нескольких вычислительных узлов, каждый со своей собственной памятью, объединённых высокоскоростными каналами связи, представляющая с точки зрения пользователя единый аппаратный ресурс.

Учитывая практическую важность задач глобальной оптимизации функций многих переменных, существующие проблемы эффективности

решения таких задач и современное состояние вопроса, исследования по разработке распределенной системы решения задач глобальной оптимизации представляется важным и актуальным.

Цель выпускной квалификационной работы магистра состоит в создании распределенного решателя задач глобальной оптимизации функции многих переменных. Для достижения цели были поставлены следующие **задачи.**

1. Провести обзор программных инструментов для решения задач глобальной оптимизации и систем построения распределенных вычислений с целью их пригодности для реализации распределенного решателя задач глобальной оптимизации.
2. Разработать распределенный решатель задач глобальной оптимизации.
3. Провести вычислительные эксперименты для тестирования вычислительной корректности и выяснения условий эффективного использования решателя задач глобальной оптимизации.

Методы, использованные в работе: методы интервального анализа, методы глобальной оптимизации, методы объектно-ориентированного программирования, методы распределенных параллельных вычислений.

Объектом работы являются параллельные алгоритмы глобальной оптимизации.

Предметом работы являются способы распределенной реализации алгоритмов глобальной оптимизации.

Апробация работы. Результаты работы докладывались на Третьей региональной конференции «Мой выбор наука!» в секции «Интеллектуальный анализ данных. Информационные системы и технологии» (г. Барнаул, апрель 2016).

Работа состоит из введения, трех глав, заключения, списка литературы и одного приложения. В работе содержится 26 рисунка, 18 листингов, 3 таблицы. Общий объем работы составляет 81 страница. Список литературы содержит 34 источника.

В первой главе рассмотрены задачи оптимизации, интервальные методы глобальной оптимизации, программные средства интервальной глобальной оптимизации и инструменты построения распределенных вычислительных систем.

Во второй главе описаны требования к распределенному решателю задач глобальной оптимизации, рассмотрена архитектура программного продукта GOptimum и механизмы фреймворка распределенных вычислений Apache Ignite, описана реализация распределенного решателя задач глобальной оптимизации.

В третей главе проведено тестирование разработанного распределенного решателя задач глобальной оптимизации на корректность решения задачи оптимизации и на эффективность использования.

1. Математические и программные средства глобальной оптимизации

1.1 Задачи оптимизации

Дадим общую постановку задачи оптимизации. Пусть $\boldsymbol{X}$ – некоторое множество (которое может совпадать со всем пространством $\mathbb{R}^n$ или же быть задано какими-то ограничениями – равенствами, неравенствами и т.п.). $\boldsymbol{f}$ – заданная на $\boldsymbol{X}$ функция, называется целевой функцией. Требуется найти приближение

$$f^* = \min_{x \in X} f(x). \tag{1.1}$$

Так формулируется задача минимизации в классической (точечной) постановке.

Методом (алгоритмом) решения задачи минимизации называют способ построения последовательности точек из $\boldsymbol{X}$, сходящихся к некоторой точке, в которой значение (1.1) точно или приближенно достигается. Типы сходимости указанной последовательности могут быть различными – от сходимости по значению функции $\boldsymbol{f}$ до сходимости с некоторой вероятностью. Поскольку на практике количество вычислительной работы всегда ограничено, то ограничиваются конечным числом членов указанной последовательности и пытаются их построить так, чтобы с наименьшими (или ограниченными) вычислительными затратами достичь желаемой (соответственно, максимально возможной) точности [5].

Сложность задачи оптимизации определяется свойствами целевой функции на интересующей нас области определения (допустимом множестве определения). Имеет место определенная дуальность свойств области $\boldsymbol{X}$ и функции $\boldsymbol{f}$. Если приходится работать со сложной функцией $\boldsymbol{f}$, то задачу можно переформулировать так, что функция станет простой, но при этом сложности будут перенесены на область определения. Возможна обратная ситуация. Чаще всего рассматривают задачи с достаточно простой областью определения, а все

сложности переносятся на целевую функцию. Одним из основных факторов, определяющих сложность задачи оптимизации, является размерность множества $\boldsymbol{X}$. В основном развиты методы решения одномерных задач, однако для большинства прикладных задач более важным является решения задач оптимизации для многомерных функций [5,7].

Существует множество различных классификаций классических методов оптимизации: по критерию размерности допустимого множества, по виду целевой функции и допустимого множества, по требованию к гладкости и наличию у целевой функции частных производных и т.д. С точки зрения вычисления производных целевой функции методы группируются следующим образом.

1. Методы нулевого порядка. Такие методы в процессе оптимизации используют только значения функции в области значений и значения аргумента в области определения; они не используют значения производных первого, второго и n-го порядков. Примеры методов нулевого порядка: метод «золотого сечения», метод Хука-Дживса, генетический алгоритм и т.д. Все эти методы отлично изложены в источниках [8].
2. Методы первого порядка (градиентные). Данные методы требуют существования первой производной оптимизируемой функции в аналитическом или численном приближённом виде. Данная группа методов использует информацию о направлении спуска к минимуму по антиградиенту, не настраивая при этом величину шага. Примеры методов первого порядка: градиентный метод с фиксированным шагом, метод наискорейшего спуска по антиградиенту, и т.д. [8].
3. Методы второго порядка (ньютоновские методы). Данные методы требуют существования первой и второй производной оптимизируемой функции в аналитическом или численном виде. Эта группа методов использует информацию о направлении спуска к минимуму по антиградиенту и

информацию о выпуклости функций, настраивая при этом величину шага [8].

Если функция не дифференцируема, но является унимодальной, выпуклой и непрерывной, то возможно применение методов нулевого порядка. Методы оптимизации первого и второго порядков очень медленно сходятся вблизи окрестности оптимума, так как или градиент близок к нулю, или матрица вторых производных плохо обусловлена. Кроме того, данные методы сходятся не к глобальному оптимуму, а к ближайшему локальному, что не допустимо при решении многоэкстремальных задач ввиду существенной погрешности вычислений. Однако, методы локальной оптимизации предоставляют базовые приемы, которые служат каркасом для решения задач глобальной оптимизации.

Идея большинства классических методов глобальной оптимизации заключается в том, чтобы тем или иным способом оценить значения целевой функции $f(\cdot)$ на некотором, по-возможности наиболее представительном, подмножестве точек из допустимого множества, и, фактически, различие методов заключается в способах выбора этих точек. Так как нам не известно, где именно на области определения можно найти глобальные оптимумы, необходимо использовать некоторую стратегию распределения этих точек [9].

Все методы многоэкстремальной оптимизации можно условно разделить на две группы детерминированные и стохастические. С помощью детерминированного алгоритма можно отыскать оценку глобального оптимума сравнив значения целевой функции в разных точках области определения. При этом достоверность результата естественном образом зависит от количества и положения выбранных точек. Основная идея детерминистских методов так или иначе состоит в получении глобального решения посредством исчерпывающего поиска на всем допустимом множестве. Поэтому с возрастанием размерности задачи такие методы теряют свою эффективность и надежность. Это является основным недостатком детерминированных алгоритмов глобальной оптимизации. На рис 1.1. приведены примеры функций, для которых глобальный

минимум нельзя найти иначе, чем путём перебора её значений на достаточно мелкой сетке [5].

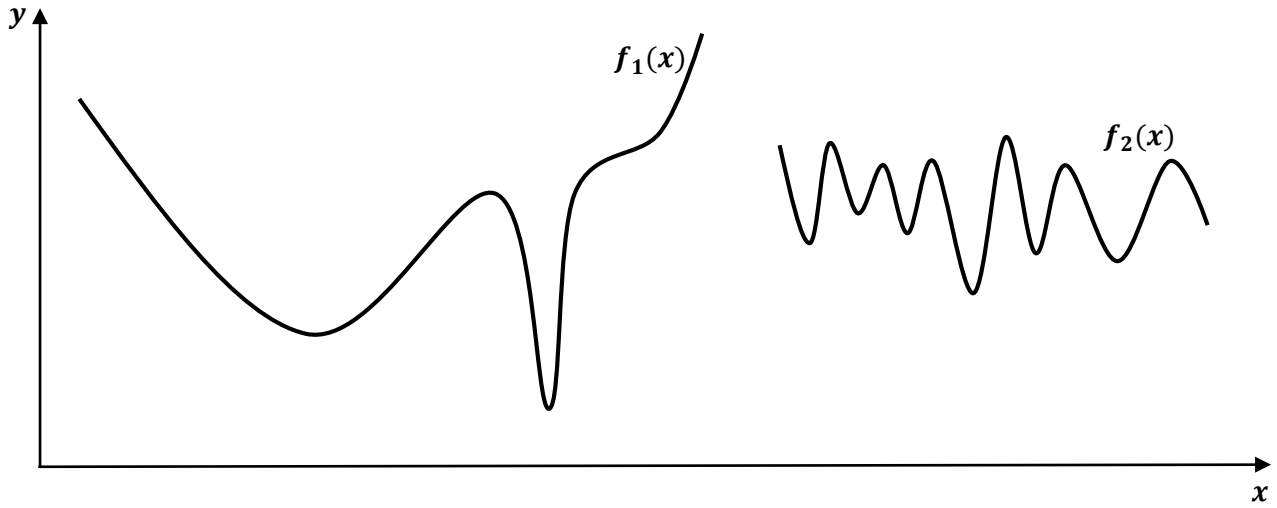

Рис. 1.1. Примеры функций, для которых трудно найти глобальный оптимум

Среди классических детерминированных методов глобальной оптимизации можно выделить методы, основанные на редукции размерности задачи. При одном из подходов многомерная задача сводится к серии вложенных одномерных задач. Другие алгоритмы используют отображения многомерной области на отрезок с помощью разверток на основе кривых Пеано [10,11]. Также существуют методы, опирающиеся на нахождение кусочно-линейных выпуклых и вогнутых опорных функций, развиваемые О.В. Хасимовым [12].

Стохастические алгоритмы в какой-то степени позволяют уйти от проблем детерминированных алгоритмов. Одним из популярных принципов организации стохастических методов является оценивание значения функции цели в случайных точках допустимого множества с последующей обработкой получившейся выборки. Как следствие, стохастические методы не гарантируют, что будет найден именно глобальный оптимум. Примерами таких методов могут служить: метод Монте-Карло, метод рандомизации, метод проб и ошибок и т.д [5].

Таким образом, можно сделать вывод, что классические алгоритмы глобальной оптимизации не всегда могут удовлетворить потребности при решении задач глобальной оптимизации. Это связано с тем, что для решения задачи доказательной (гарантированной) глобальной оптимизации, когда

требуется не только найти оптимум функции, но и дать гарантию того, что найденное решение является действительно глобальным экстремумом, классические стохастические алгоритмы не подходят, а классические детерминистские требуют очень больших трудозатрат. Поэтому рассмотрим еще один тип методов – интервальные методы глобальной оптимизации.

1.2 Интервальные методы глобальной оптимизации

В настоящем разделе описываются интервальные методы глобальной оптимизации. Приведенное описание, в основном, следует [5], [6], [7].

Постановка задачи глобальной оптимизации с использованием интервальных методов выглядит следующим образом:

Дана вещественнозначная функция $f\colon \mathbb{R}^n \supset X \to \mathbb{R}$ на некоторой подобласти $\boldsymbol{x}$ её области определения X. При этом $\boldsymbol{x}$ представляет собой прямоугольный брус, с параллельными координатным осям сторонами:

$$\text{найти} \min_{x \in \boldsymbol{x}} f(x). \tag{1.2}$$

В случае, когда нам априори не известен характер поведения целевой функции f и структура её локальных экстремумов, решение задачи (1.2) неизбежно потребует в том или ином виде перебор и сравнение «всех точек» области определения.

Наилучшим решением будет найти некие x' и y', наиболее точно приближающие точный результат. То есть такие, что расстояния $|x' - x^*|$ и $|y' - y^*|$ будут «малы». Таким образом, мы получаем следующую постановку задачи.

Для функции $f(x)$, заданной на некоторой области определения $x_0 \le x \le x_1$, необходимо найти $\boldsymbol{X} \coloneqq \left[\underline{\boldsymbol{x}}, \overline{\boldsymbol{x}}\right]$ и $\boldsymbol{Y} \coloneqq \left[\underline{\boldsymbol{y}}, \overline{\boldsymbol{y}}\right]$, такие что:

1) $x' \in \boldsymbol{X}$ и $y' \in \boldsymbol{Y}$ и
2) $\boldsymbol{X}$ и $\boldsymbol{Y}$ настолько узки, насколько это нужно для данной задачи.

Таким образом, основной принцип итерационных интервальных методов глобальной оптимизации следующий. Так или иначе, все методы этого типа существенно эксплуатируют тот факт, что большинство интервальных оценок области значений функции – асимптотически точные, т.е.

$$\text{dist}\,(\boldsymbol{f}(\boldsymbol{x}), \text{range}_{\boldsymbol{x}} f) \to 0 \qquad \text{при wid}\,(\boldsymbol{x}) \to 0, \tag{1.3}$$

где $\boldsymbol{f}(\boldsymbol{x})$ – интервальное расширение функции f, $\text{range}_{\boldsymbol{x}} f = \{f(x) \mid x \in \boldsymbol{x}\}$ – область значений функции f на брусе $\boldsymbol{x}$, $\text{wid}\,(\boldsymbol{x})$ – ширина бруса.

Интервальным расширением функции согласно [6] является внешняя интервальная оценка области значения $\text{range}_{\boldsymbol{x}} f$.

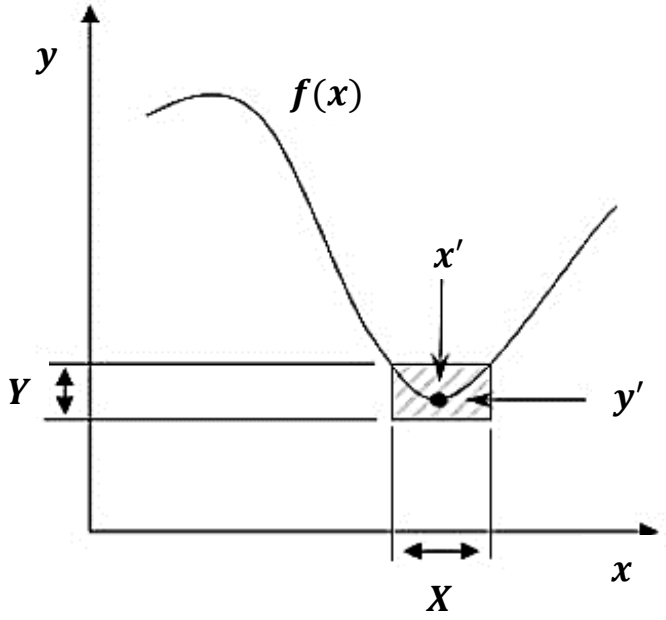

Рис. 1.2.Иллюстрация интервальной постановки задачи.

При уменьшении размеров области определения (бруса) точность интервального расширения функции увеличивается. К сожалению, при этом не следует ожидать, что при уменьшении размеров области определения, скажем вдвое, и точность интервальной оценки улучшится на столько же, или хотя бы пропорционально. Это лишь означает, что при принудительном дроблении бруса области определения точность интервальной оценки будет возрастать.

Если мы разобьем исходный брус $\boldsymbol{x}$ на два подбруса $\boldsymbol{x}'$ и $\boldsymbol{x}''$, дающие в

объединении весь $\boldsymbol{x}$, то есть такие, что $\boldsymbol{x}' \cup \boldsymbol{x}'' = \boldsymbol{x}$, то

$$\{ f(x) | x \in \boldsymbol{x} \} = \{ f(x) | x \in \boldsymbol{x}' \} \cup \{ f(x) | x \in \boldsymbol{x}'' \}.$$

Соответственно, можно вычислить интервальное расширение на каждом подбрусе и в качестве новой оценки минимума целевой функции на $\boldsymbol{x}$ взять

$$\min\{ \underline{\boldsymbol{f}(\boldsymbol{x}')}, \underline{\boldsymbol{f}(\boldsymbol{x}'')} \}$$

и она будет более точна, чем исходная оценка $\boldsymbol{f}(\boldsymbol{x})$, так как у брусов $\boldsymbol{x}'$ и $\boldsymbol{x}''$ размеры меньше, чем у исходного $\boldsymbol{x}$. В свою очередь брусы-потомки $\boldsymbol{x}'$ и $\boldsymbol{x}''$ можно разбить на более мелкие части, найти для них интервальные расширения и затем уточнить оценку для минимума, потом повторить процедуру и так далее по алгоритму, примерно как на рис. 1.3.

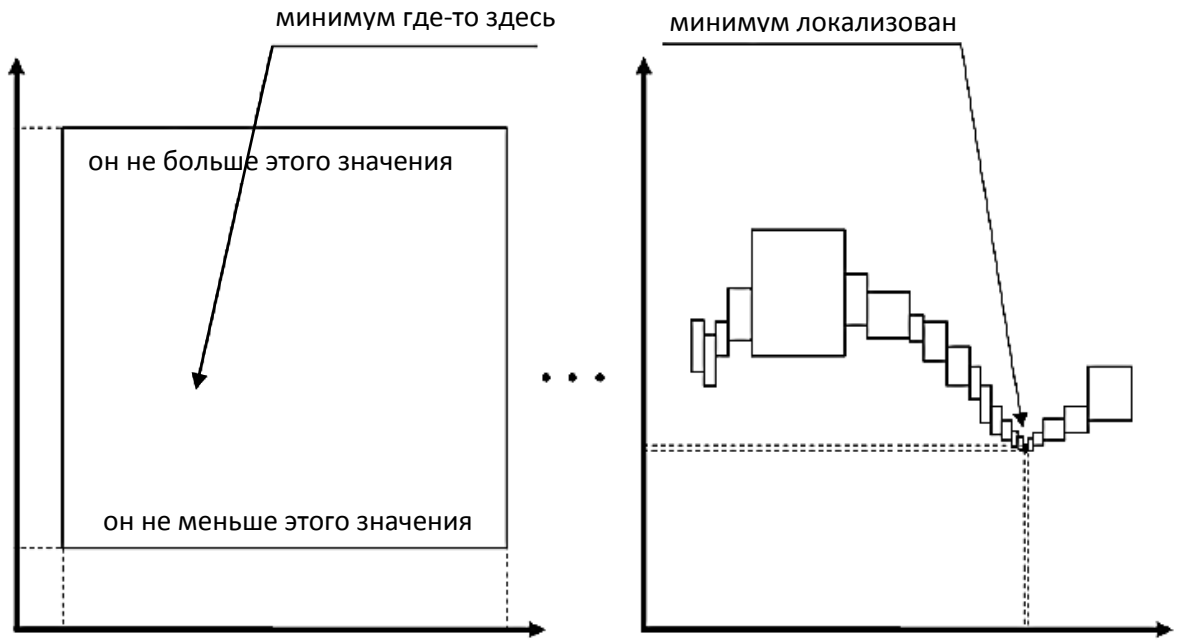

Рис. 1.3. Поиск оптимума последовательным дроблением области определения.

Целесообразно внести некоторый порядок в этот процесс последовательного дробления-оценивания. Самый разумный выбор – дробить брус с наименьшей границей, (с наибольшей границей если нас интересует максимум целевой функции). Существует несколько методов дробления бруса несколько. Первый метод − это дробление большей стороны, когда за одну

итерацию алгоритма выполняется одно деление, при этом делится наиболее широкая сторона бруса. Второй метод – это дробление бруса сразу по всем координатам. К примеру, для функции от двух переменных за одну итерацию алгоритма брус будет разделен сразу на четыре части. Третий метод – это эвристическое дробление. Суть метода в том, что сторона для очередного дробления выбирается исходя из результативности предыдущих дроблений. Каждый раз, производя деление очередного бруса, алгоритм анализирует, насколько улучшилась интервальная оценка, при делении именно по этой координате. На рис. 1.4 приведена блок-схема широко распространённого метода «адаптивного интервального дробления».

Алгоритм также оперирует понятием «рабочий список» – это список (реализуемый или как список, или стек или куча), где хранятся все подбрусы, порожденные алгоритмом. Ведущий или наиболее перспективный брус – это брус, на котором в настоящий момент достигается наибольшая (наименьшая) оценка значения функции. Критерием остановки работы алгоритма может быть максимальное количество итераций, достаточная ширина оценки оптимума.

Конечно, приведенный простейший алгоритм глобальной оптимизации едва ли может быть с успехом применен к решению серьезных практических задач. Фактически, при уменьшении размеров области, подозрительной на глобальный оптимум, основной упор в нем делается на дробление пополам (бисекцию), эффект от которого при увеличении размерности n становится все менее и менее ощутимым. Основное достоинство этого алгоритма − простота реализации и гарантированность результатов.

Кроме простого хранения брусов, алгоритм вынужден выполнять определенные действия по поддержанию рабочего списка упорядоченным либо же по поиску наиболее перспективного бруса при неупорядоченном хранении. Существует несколько критериев, с помощью которых можно выделять наиболее перспективные брусы или же наоборот отбраковывать бесперспективные.

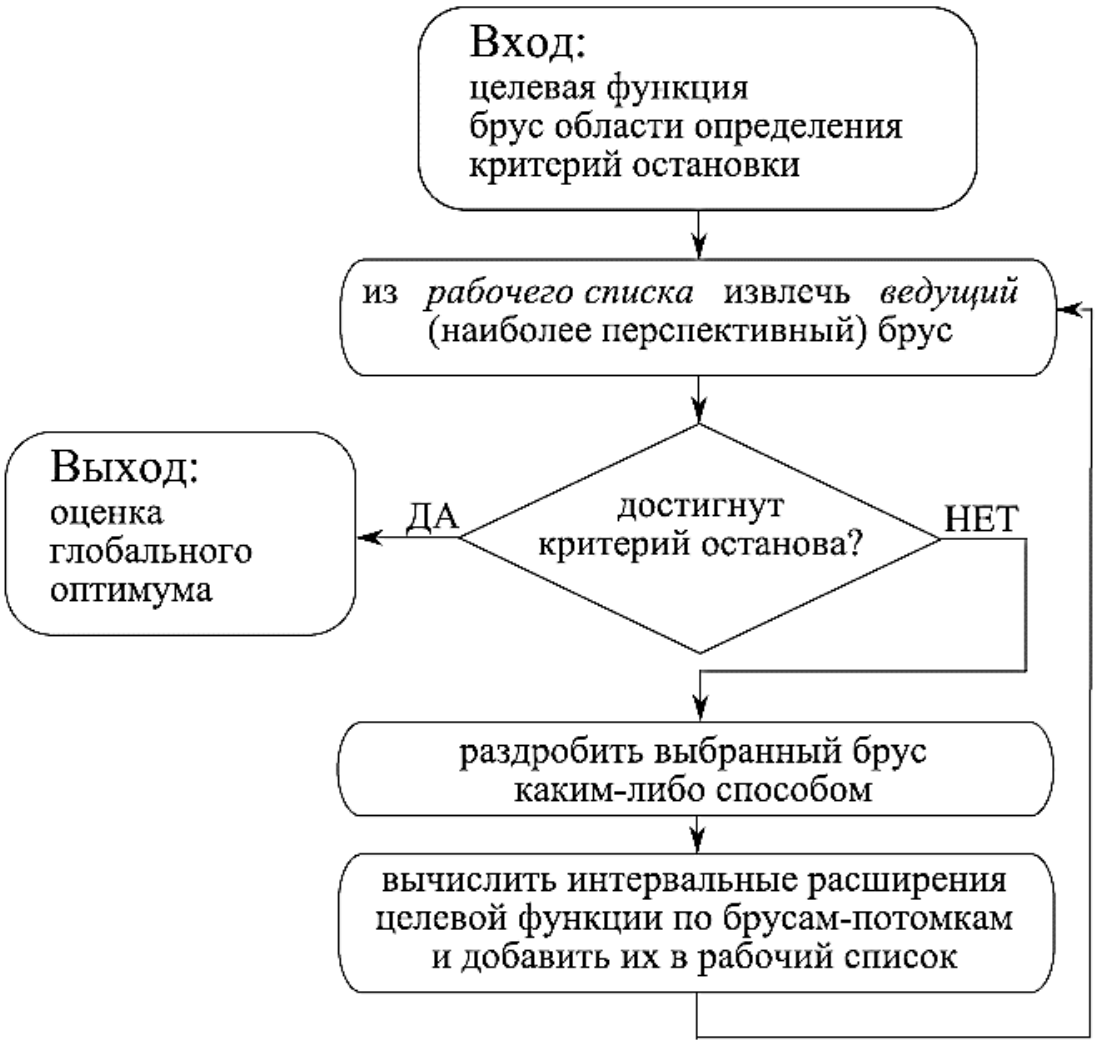

Рис. 1.4. Блок-схема оптимизационного алгоритма адаптивного дробления.

Отбраковка по значению

Можно использовать в качестве порогового значения верхнюю границу ведущего бруса. Например, в результате работы алгоритма оптимизации получилась конфигурация подобная показанной на рис. 1.5.

Начнем с того, что интервальное расширение функции гарантирует, что значения функции на данном интервале не выйдут за обозначенные границы. Как видно из рисунка брусы с номерами 1, 3, 7, 8, 9 гарантированно не содержат глобальный минимум, поскольку нижняя грань их значений находится выше верхней грани одного из брусов. А это значит, что нижняя оценка значений функции на этих брусах больше, чем верхняя оценка значений функции на брусе номер 5. То есть, какие бы значения не принимала целевая функция внутри бруса

номер 5, она всё равно будет меньше любого своего значения на брусах 1, 3, 7, 8, 9.

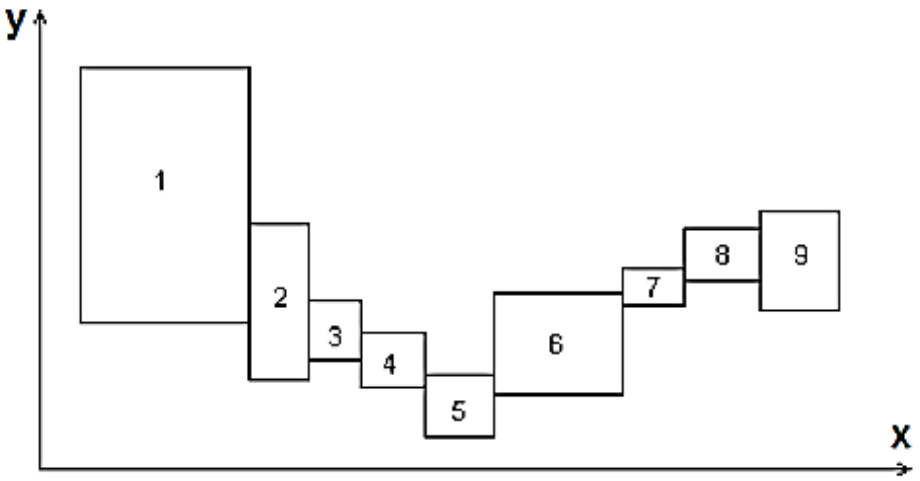

Рис.1.5. Пример конфигурации брусов в процессе оптимизации

Преимущество такого выбора состоит лишь в отсутствии каких-либо дополнительных вычислений: верхняя граница уже известна, но оценка верхней границы интервального расширения может быть избыточной. Повысить эффективность метода можно, вычислив точное значение целевой функции, например, в середине интервала.

Тест на монотонность

В точке экстремума производная по каждой переменной должна обращаться в нуль. Таким образом, если на каком-либо подбрусе интервальное расширение первой производной целевой функции по какой-либо координате не содержит нуля, то такой брус гарантированно не содержит точек экстремума, а, следовательно, может быть безболезненно удален из рассмотрения. Единственная оговорка состоит в том, что тест на монотонность нельзя применять к граничным брусам, так как оптимум может достигаться на границе и при этом может быть не равен нулю.

Тест на выпуклость-вогнутость

Мы можем с помощью второй производной функции, которая содержит информацию о форме графика – выпуклости или вогнутости функции вблизи этой точки, а также интервального расширения второй производной,

позволяющего узнать вид функции на целом брусе, отсеивать брусы не той кривизны, то есть вогнутые, если нас интересует глобальный минимум и выпуклые, если наша цель – глобальный максимум.

Интервальный метод Ньютона

Интервальная версия итерационного численного метода решения нелинейных уравнений, в котором поиск решения осуществляется путём построения последовательных приближений. Интервальный метод Ньютона позволяет локализовать корни, в том числе нелинейных уравнений. В задачах глобальной оптимизации это может быть использовано при применении методов распространения ограничений, а также при прямом поиске нулей первой производной целевой функции. В точке экстремума частные производные должны обращаться в ноль. И как уже говорилось, если на каком-либо неграничном подбрусе интервальное расширение первой производной целевой функции по какой-либо переменной не содержит нуля, то он гарантированно не содержит точек экстремума, и может быть удален из рассмотрения.

Таким образом, оставляя наиболее перспективные брусы и отбраковывая неперспективные, алгоритм дробит интересующую нас область на все более мелкие подобласти и, соответственно, увеличивая точность оценивания можно найти значение глобального оптимума. Но грубость интервальной оценки, а также возможное ее застаивание, могут существенно ухудшить работу всего алгоритма. То есть это та ситуация, когда достаточно большое количество последовательных дроблений не приводит к значительному улучшению точности интервальной оценки. Дополнительно усугубляют дело «ложные оптимумы», то есть брусы, признаваемые ведущими, хотя реально и не содержащие глобального оптимума. Всё это приводит к сильному раздутию рабочего списка.

Один из возможных выходов из создавшегося положения – изменение процедуры выбора ведущего бруса введением случайности (рандомизации) в этот процесс. Идея отказаться от жёсткого детерминизма классических методов интервальной глобальной оптимизации, с целью получения гибридных

алгоритмов, которые позволяют производить гарантированную оценку глобального оптимума и в тоже время при просмотре рабочего списка случайным образом выбирать ведущий брус, была предложена С.П. Шарым в публикациях [7,13,14,15] и Н.В. Пановым в публикациях [5,14,15]. Рассмотрим примеры таких алгоритмов.

1. Случайное интервальное дробление.

Алгоритм случайного интервального дробления является простейшим из методов, сочетающих интервальную технику оценивания значений функции со стохастическим управлением. Основное отличие такого алгоритма от классического интервального алгоритма в том, что ведущий брус извлекается из рабочего списка случайным образом. Основным недостатком такого алгоритма является то, что точность оценки глобального оптимума довольно быстро выходит на асимптоту и при дальнейших итерациях значительно не улучшается. Алгоритм не тратив время на упорядочивание рабочего списка или поиска наиболее перспективного бруса, а лишь дробит брусы, как следствие он может быстро погрязнуть в порожденном множестве подбрусов.

2. Случайное интервальное дробление с приоритетом.

Алгоритм случайного интервального дробления с приоритетом является доработанным алгоритмом случайного интервального дробления. Единственное, но весьма существенное отличие − это модификация процедуры выбора очередного кандидата на дробление. Выбор по-прежнему происходит случайно, но более широкие брусы имеют большую вероятность быть раздробленными.

3. Интервальный алгоритм имитации отжига.

За основу алгоритма взят классический (точечный) метод (известный также как алгоритм Метрополиса [16,17,18]), моделирующий физические процессы отжига или кристаллизации и хорошо зарекомендовавший себя для многоэкстремальных проблем с большим количеством возможных решений [17,18]. Алгоритм оперирует таким понятием как «температура». Чем она выше, тем больше вероятность того, что на определенном шаге будет выбран брус, не доставляющий рекордную на данный момент оценку оптимум, то есть, тем

больше «рысканье» алгоритма по области определения. По мере работы алгоритма температура понижается и «рысканье» прекращается.

4. Интервальный генетический алгоритм.

Алгоритм представляет собой схему, которая в своей основе является генетическим алгоритмом, но на этапах оценивания использует интервальный инструментарий.

5. Мультиметодный алгоритм.

Основная идея мультиметодного алгоритма следующая. Запускается любой алгоритм глобальной оптимизации. Через короткий промежуток времени он останавливается и следом запускается следующий алгоритм и так далее на ходу подменяем алгоритмы оптимизации. При этом отслеживаем относительную эффективность каждого из алгоритмов – сколько итерации было сделано, насколько улучшилась оценка оптимума, как изменился размер рабочего список и т.п. Сравнивая успешность методов, можно динамически регулировать квант времени, выделяемый каждому из алгоритмов, обеспечивая оптимальность применения методов.

Все рассмотренные методы детально описаны в публикации Н.В. Панова [5].

Таким образом, использование стохастических интервальных методов позволяет конструировать для решения задач глобальной оптимизации вычислительные технологии, хорошо справляющиеся со многими практическими задачами, но вычислительная эффективность, которая обеспечивается интервальными методами глобальной оптимизации, такова, что даже на многоядерном вычислительном узле, можно решать задачи ограниченного класса за приемлемое время. Для уменьшения времени поиска глобального оптимума есть резерв в виде распараллеливания решения этой задачи на распределенную систему, которое можно использовать естественным образом, ввиду вычислительной независимости задачи.

1.3 Программные средства интервальной глобальной оптимизации

Рассмотрим вопрос о том, какие программные инструменты можно использовать на практике, пользователю, который захотел решить прикладную задачу. Существует определенный набор таких инструментов, у каждого есть свои достоинства и недостатки. Рассмотрим некоторые из них поподробнее.

BARON Software

BARON [19] представляет собой вычислительную систему для глобального решения задач нелинейного программирования, а также смешанного целочисленного нелинейного программирования. В то время как упомянутые задачи обеспечивают глобальные оптимальные решения только при определенных предположениях о выпуклости, BARON может работать с детерминированными глобальными алгоритмами оптимизации по типу алгоритмов метода ветвей и границ, которые гарантированно обеспечивают глобальные оптимальные решения при достаточно общих предположениях.

BARON использует алгоритмы метода ветвей и границ, усиленные различными методами распространения ограничений и двойственности для уменьшения диапазонов переменных в ходе решения алгоритма.

BARON может обрабатывать вещественные нелинейные функции, которые включают $\exp(x), \ln(x)$. В настоящее время не существует методов решения для других функций, включая тригонометрические функции $\sin(x), \cos(x)$ и т.д.

Система BARON представляет собой приложение, с закрытым исходным кодом. Она содержит язык моделирования высокого уровня, позволяющий считывать смешанные целочисленные модели нелинейной оптимизации в относительно простом формате. Ввод осуществляется в виде текстового файла (см. рис.1.6).

p2.bar

```
MODULE: NLP;
POSITIVE_VARIABLE x;
LOWER_BOUND{
x: 1;
}
UPPER_BOUND{
x: 2;
}
OBJ: minimize  -500 * x + 2.5 * x^2 + 1.666666666 * x^3 +
               1.25 * x^4 + 1 * x^5 + 0.8333333 * x^6 +
               0.714285714 * x^7 + 0.625 * x^8 + 0.555555555 * x^9 +
               x^10 -43.636336 * x^11 + 0.41666666 * x^12 +
               0.384615384 * x^13 + 0.357142857 * x^14 + 0.3333333 * x^15 +
               0.3125 * x^16 + 0.294117647 * x^17 + 0.277777777 * x^18 +
               0.263157894 * x^19 + 0.25 * x^20 + 0.238095238 * x^21 +
               0.227272727 * x^22 + 0.217391304 * x^23 + 0.208333333 * x^24 +
               0.2 * x^25 + 0.192307692 * x^26 + 0.185185185 * x^27 +
               0.178571428 * x^28 + 0.344827586 * x^29 + 0.6666666 * x^30
               -15.48387097 * x^31 + 0.15625 * x^32 + 0.1515151 * x^33 +
               0.14705882 * x^34 + 0.14285712 * x^35 + 0.138888888 * x^36 +
               0.135135135 * x^37 + 0.131578947 * x^38 + 0.128205128 * x^39 +
               0.125 * x^40 + 0.121951219 * x^41 + 0.119047619 * x^42 +
               0.116279069 * x^43 + 0.113636363 * x^44 + 0.1111111 * x^45 +
               0.108695652 * x^46 + 0.106382978 * x^47 + 0.208333333 * x^48 +
               0.408163265 * x^49 + 0.8 * x^50 ;

```

Normal text file | length : 1264 | lines : 26 | Ln : 1 | Col : 1 | Sel : 0 | 0 | UNIX | UTF-8 | INS

Рис.1.6. Пример входного файла приложения BARON

```
 For NLP: IPOPT, FILTERSD
================================================================================
 Starting solution is feasible with a value of     36.1767610000
 Doing local search
 Solving bounding LP
 Starting multi-start local search
 Preprocessing found feasible solution with value  4.23791612465
 Done with local search
================================================================================
  Iteration    Open nodes         Time (s)    Lower bound       Upper bound
*         1             0             0.02     1.00000          1.00006
          1             0             0.02     1.00000          1.00006

 Cleaning up

                       *** Normal completion ***

 Wall clock time:                    0.02
 Total CPU time used:                0.02

 Total no. of BaR iterations:       1
 Best solution found at node:       1
 Max. no. of nodes in memory:       1

 All done
================================================================================
```

Рис.1.7. Пример выходного файла приложения BARON

Несмотря на то, что это не является жестким требованием, настоятельно рекомендуется, чтобы все входные файлы BARON имели расширение .bar. В результате работы приложения выдается ответ с решением задачи (см. рис.1.7).

Основными недостатками системы BARON являются закрытость ее исходного кода, из-за чего не имеется возможности каким-либо образом ее модифицировать, и трудности с запуском на современных операционных системах.

GlobSol

GlobSol [20] является самостоятельной библиотекой реализованной на языке Fortran 90/95 с открытым исходным кодом для нахождения достоверных решений задач глобальной оптимизации, с ограничениями и без ограничений, а также для нахождения точных интервалов для всех решений систем нелинейных уравнений. GlobSol использует интервальный метод ветвей и границ. Таким образом, структура базового алгоритма GlobSol является схожей структурам алгоритмов многих других пакетов оптимизации, таких как упомянутый выше BARON Software.

Тем не менее, цель GlobSol заключается в обеспечении (поиске) строгих границ для всех возможных решений. То есть, выходные данные GlobSol представляют собой список интервальных оценок (брусов), за пределами которых GlobSol с математической строгостью доказал, отсутствие глобального оптимума. Это делается путем тщательной разработки алгоритмов и использования направленного округления. Таким образом, GlobSol хорошо подходит для решения задач, в которых гарантированность оценок решений желательна или важна. В задачах, где такие гарантии не обязательны, другое программное обеспечение может решить их более эффективно, или может решать более широкий круг задач.

Существует параллельная реализация ParaGlobSol [21] этой библиотеки. Она представляет собой параллельную систему глобальной оптимизации, которая была разработана группой исследователей из США и Китая. Параллельная реализация также написана на языке Fortran 90/95 с используемым MPI для обеспечения межпроцессного взаимодействия. Система реализована в двух версиях ParaGlobSol 0.1 и 0.2. Обе версии были запущены на сетевых

станциях SUN Ultra Station, и успешно решили все тестовые задачи, которые были решены с помощью последовательной системы GlobSol.

```
PROGRAM SIMPLE_MIXED_CONSTRAINTS

  USE CODELIST_CREATION

! The independent variables are in the CDLVAR array passed
! to INITIALIZE_CODELST.  The number of independent variables
! must be the exact dimension of this array.
   TYPE(CDLVAR), DIMENSION(2):: X
! The objective function must be declared this way.
   TYPE(CDLLHS):: PHI
! The dimension of this array should be the number of inequality
! constraints.
   TYPE(CDLINEQ), DIMENSION(2):: G
! The dimension of this array should be the number of equality
! constraints.
   TYPE(CDLEQ), DIMENSION(1) :: C

      CALL INITIALIZE_CODELIST(X)

! This statement defines the objective
  PHI = -2*X(1)**2 - X(2)**2

! These two statements define the inequality constraints
! G(1) <= 0 and G(2)<= 0.
  G(1) = X(1)**2 + X(2)**2 - 1
  G(2) = X(1)**2 - X(2)

! This statement defines the equality constraint C(1) == 0.
  C(1) = X(1)**2 - X(2)**2

      CALL FINISH_CODELIST

END PROGRAM SIMPLE_MIXED_CONSTRAINTS
```

Рис.1.8. Пример программы использующей библиотеку GlobSol

Для того, чтобы реализации параллельного алгоритма оказались более эффективными, были испробованы разные схемы, чтобы сбалансировать нагрузку. ParaGlobSol 0.1 уравновешивает нагрузку равномерно, распределяя изначальные брусы на все процессоры, в то время как ParaGlobSol 0.2 динамически уравновешивает нагрузку, распределяя за раз один брус на процесс, который этого требует.

Приложения GlobSol и его параллельная модификация ParaGlobSol имеют следующий недостаток. Поскольку основной метод поиска глобального оптимума реализованный, в приложении представляет собой алгоритм ветвей и границ, то порядок перебора рабочего списка никак не зависит от структуры

функции. Так как возможно подобрать такие практические задачи, когда наилучшее решение будет отыскиваться на поздних шагах алгоритма поиска глобального оптимума, то время поиска оптимума будет большим. Перспективнее выглядит использование стохастических методов, которые позволяют нам «перемешивать» рабочий список, вследствие чего, появляется возможность находить решения раньше, уменьшая время поиска глобального оптимума. Поэтому рассмотрим программное обеспечение реализующее стохастические методы интервальной глобальной оптимизации.

GOptimum

Система поиска глобального оптимума GOptimum [22], разработанная в рамках диссертационного исследования Панова Н.В. [5,14,15], позволяет использовать различные алгоритмы поиска глобального оптимума на заданной многопараметрической функции. GOptimum реализован на языке Java, имеет открытый исходный код. В этой системе реализованы новые и модифицированы уже существующие интервальные алгоритмы глобальной оптимизации, основанные на привлечении стохастики (рандомизации):

- алгоритм бисекции с перекрытием;
- случайный интервальный поиск;
- случайный интервальный поиск с приоритетом;
- интервальный алгоритм имитации отжига;
- интервальный генетический алгоритм;
- адаптивный мультиметодный алгоритм;
- оптимизированный адаптивный параллельный алгоритм глобальной интервальной оптимизации;

GOptimum работает в пакетном режиме. Приложение принимает на вход файл, в котором описана задача, а именно целевая функция, переменные, заданные в интервальной форме и некоторые дополнительный опции такие как «ReportArea», «ReportValue», «MaxSteps», «Epsilon», «Optimum»,

«UseSimpleRounding», с помощью которых можно задавать параметры решения задачи и вывода ответа. Система GOptimum использует библиотеку ia_math, которая содержит реализацию арифметических и операций над интервалами [23].

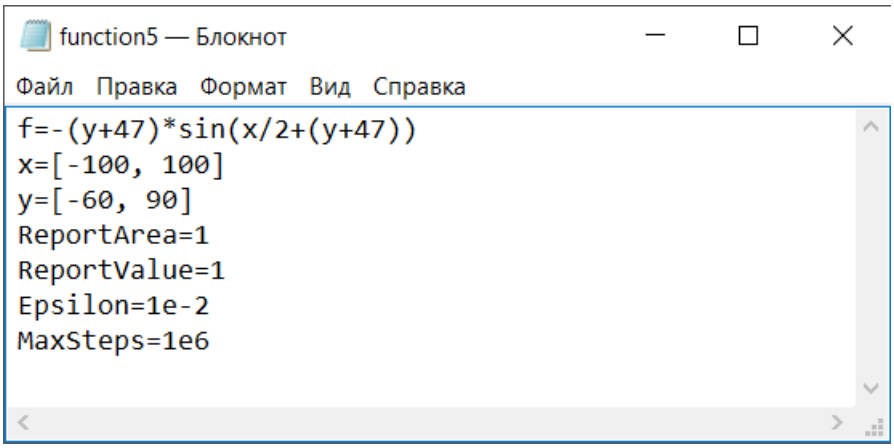

Рис.1.9. Пример входного файла приложения GOptimum

После окончания работы алгоритма, значение найденного глобального экстремума и значения, где были локализованы другие экстремумы функции, записываются в выходной файл (см. рис 1.10).

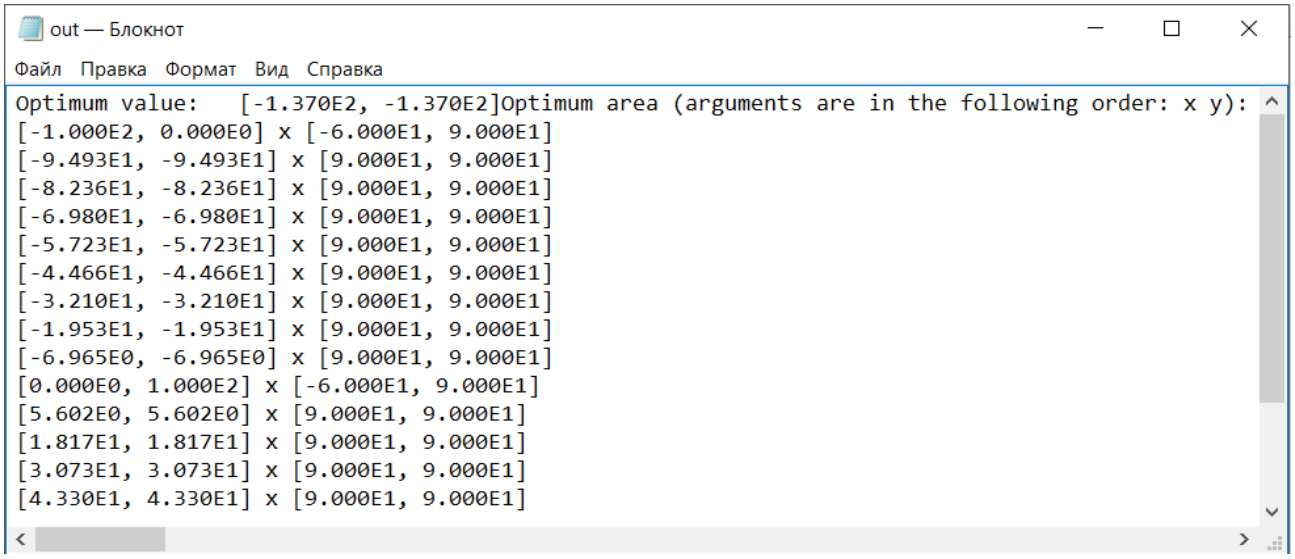

Рис.1.10. Пример выходного файла приложения GOptimum

Исходная область определения целевой функции делится на подбрусы по числу доступных процессоров на машине, каждому из них отдается свой исходный подбрус и затем запускается по независимому алгоритму глобальной

оптимизации со своей начальной областью на каждом процессоре. Во время работы алгоритмы обмениваются своими текущими верхними и нижними оценками глобального экстремума. Никакой метод не обменивается данными ни с одним другим напрямую. Если в соответствии со своей внутренней логикой какой-то метод решает, что своими текущими результатами стоит обменяться с другими методами, он просто делает эти данные публичными. Иными словами, есть некая область памяти, про которую известно всем алгоритмам. Каждый из алгоритмов может прочитать из неё текущее значение, и, если его оценки лучше, то обновить рекорд. Иначе прочитанные значения используются для процедур отбраковки брусов.

Использование многопоточной технологии параллельного вычисления ограничивает вычислительный узел пределами одной Java-машины. Если необходимо запустить поиск глобального экстремума на вычислительных системах большего масштаба, то использовать GOptimum в том виде, в котором он сейчас существует мы не можем. Однако, GOptimum имеет явные преимущества. Во-первых, приложение имеет открытый код, который можно изменять и дорабатывать. Библиотека интервальной арифметики, которая используется в GOptimum, также имеет открытый исходный код. Во-вторых, приложение реализовано на современном, развивающемся языке Java, который работает на виртуальной Java-машине, что позволяет запускать вычисления на разных платформах. В-третьих, простота свойственная реализации алгоритмов интервальной глобальной оптимизации, а также поддержка параллельного алгоритма для одного вычислительного узла, который в дальнейшем мы можем использовать для реализации распределенной системы решения задач глобальной оптимизации.

На основе всего выше изложенного, для дальнейшей разработки распределенной системы в качестве инструмента поиска глобального оптимума была выбрана система GOptimum.

1.4 Обзор распределенных вычислительных систем

Распределенная вычислительная система представляет собой программно-аппаратный комплекс, ориентированный на решение определенных задач. С одной стороны, каждый вычислительный узел является автономным элементом. С другой стороны, программная составляющая распределенной вычислительной системы должна обеспечивать пользователям видимость работы с единой вычислительной системой. В связи с этим выделяют следующие важные характеристики таких систем.

1. Возможность работы с различными типами устройств:
   - с различными поставщиками устройств,
   - с различными операционными системами,
   - с различными аппаратными платформами.

   Вычислительные среды, состоящие из множества вычислительных систем на базе разных программно-аппаратных платформ, называются гетерогенными;
2. Возможность простого расширения и масштабирования.
3. Перманентная (постоянная) доступность ресурсов (даже если некоторые элементы распределенной вычислительной системы некоторое время могут находиться вне доступа).
4. Сокрытие особенностей коммуникации от пользователей.

Для обеспечения работы гетерогенного оборудования вычислительной сети в виде единого целого, стек программного обеспечения обычно разбивают на два слоя. На верхнем слое располагаются распределенные приложения, отвечающие за решение определенных прикладных задач средствами распределенной вычислительной системы. Их функциональные возможности базируются на нижнем слое – промежуточном программном обеспечении, которое взаимодействует с системным программным обеспечением и сетевым уровнем, для обеспечения прозрачности работы приложений в распределенной системе (см. рис. 1.11) [24].

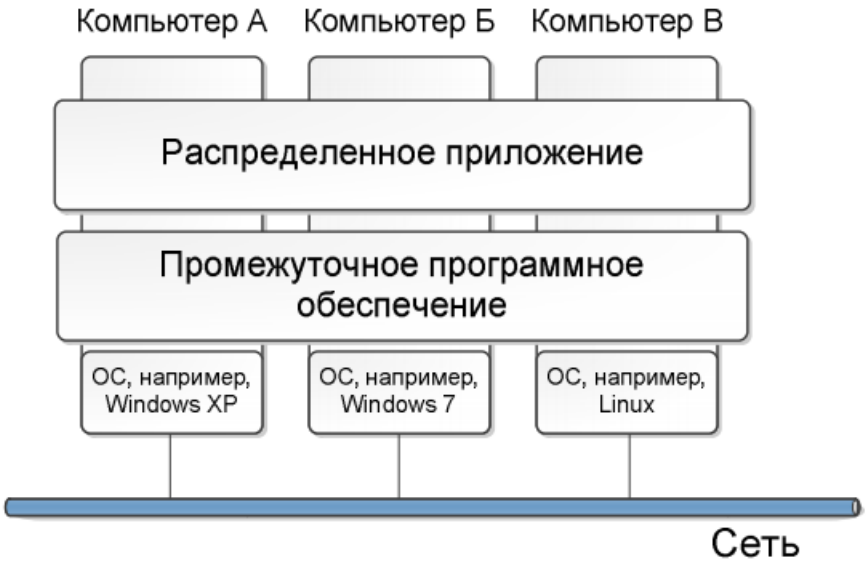

Рис.1.11. Слои программного обеспечения в распределенной вычислительной системе

Для реализации распределенного решателя задач глобальной оптимизации было рассмотрено несколько распределённых вычислительных платформ. Основным критерием выбора была поддержка языка программирования Java, а также возможность использования бесплатной версии платформы. Рассмотрим их подробнее.

Hazelcast

Hazelcast [25] − это распределенная система с открытым кодом. Она обеспечивает поддержку различных технологий, таких как In-Memory Computing, распределенная кэш-память, поддерживает разработку на многих языках программирования и обеспечивает взаимодействие со всевозможными базами данных. Hazelcast является масштабируемой системой. Распределенные приложения могут использовать Hazelcast для распределенного кэширования, синхронизации, кластеризации, обработки и т.д.

Hazelcast поставляется в виде jar-библиотеки, она легко подключается к программному обеспечению и обеспечивает распределенную систему.

Hadoop

Hadoop [26] – это свободно распространяемый набор утилит, библиотек и фреймворк для разработки и выполнения распределённых программ, работающих на кластерах. Hadoop разработан на Java в рамках вычислительной

парадигмы MapReduce, согласно которой приложение разделяется на большое количество одинаковых элементарных заданий, выполнимых на узлах кластера и естественным образом сводимых в конечный результат.

Одна из основных целей Hadoop − это обеспечение горизонтальной масштабируемости кластера посредством добавления недорогих узлов, без использования мощных серверов и дорогих систем хранения данных. Масштабируемость Hadoop-систем в значительной степени зависит от характеристик обрабатываемых данных, прежде всего, их внутренней структуры и особенностей по извлечению из них необходимой информации, и сложности задачи по обработке, которые, в свою очередь, диктуют организацию циклов обработки, вычислительную интенсивность атомарных операций, и, в конечном счёте, уровень параллелизма и загруженность кластера.

Apache Ignite In-Memory data fabric

Apache Ignite [27] − это программное обеспечение для обработки данных внутри памяти в распределенной среде. Эта платформа позволяет неограниченно масштабировать созданный кластер. Apache Ignite обеспечивает единый API, который охватывает основные типы приложений такие как Java, .Net, C++ и позволяет им взаимодействовать с множеством хранилищ, содержащих структурированные, полуструктурированные и неструктурированные данные (см. рис.1.12).

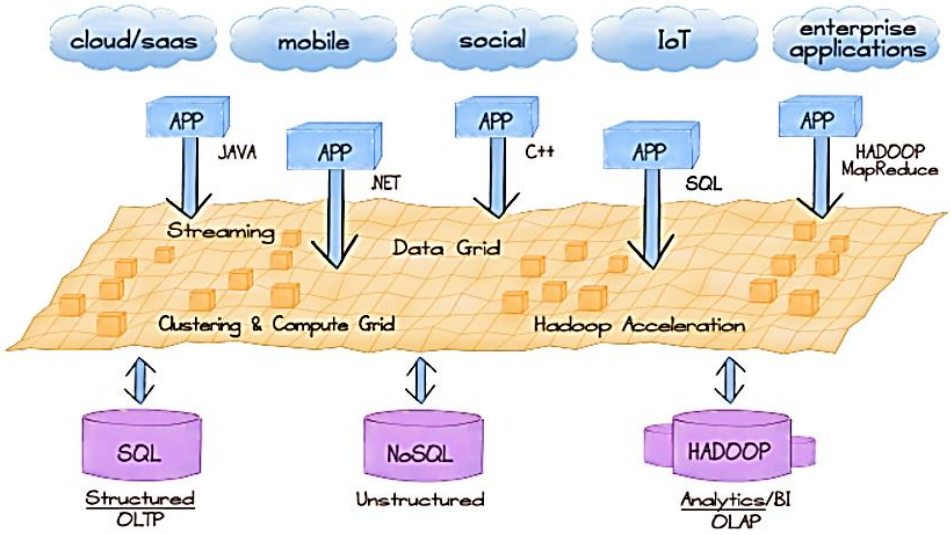

Рис.1.12. Компоненты Apache Ignite In-Memory Data Fabric

Технология In-Memory Computing реализованная в Apache Ignite предлагает стратегический подход к вычислениям в памяти узлов, который обеспечивает высокую производительность и масштабируемость.

Распределенные вычисления в Apache Ignite выполняются параллельно, чтобы получить высокую производительность, малую задержку и линейную масштабируемость. Технология Ignite compute grid обеспечивает набор простых API, которые позволяют пользователям распределять вычисления и обработку данных на нескольких узлах кластера. Распределенная параллельная обработка основана на способности браться за любые вычисления, выполнять их на любом количестве узлов и затем возвращать результаты. Вычисления на кластере могут запускаться по-разному. Например, можно запустить задания на узлах синхронно либо асинхронно. Есть возможность произвести вычисления с помощью встроенной реализации MapReduce.

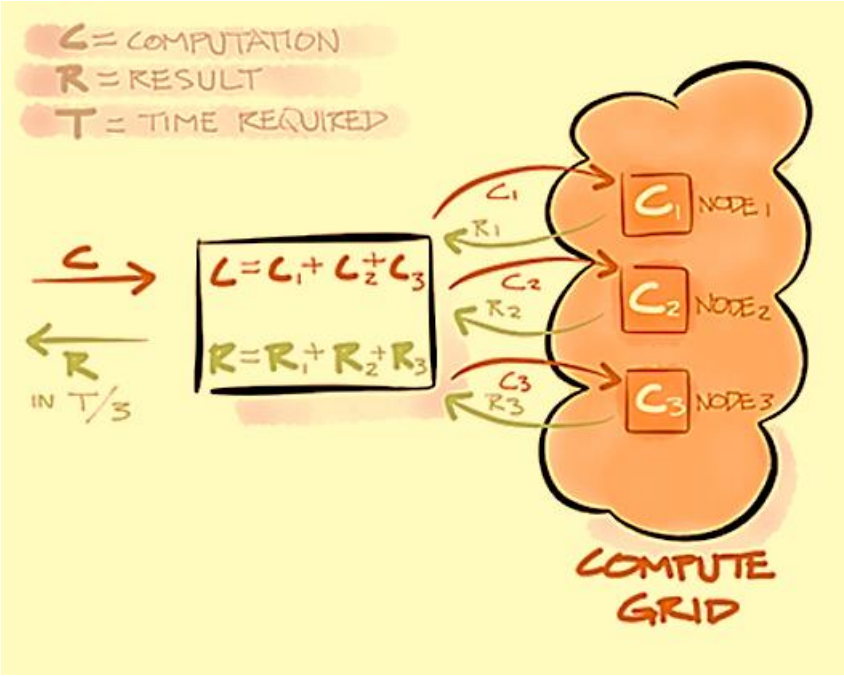

Рис.1.13. Схема работы Compute Grid

Apache Ignite является мощным инструментом для сложных вычислений на распределенной системе. Он имеет следующие преимущества:

1. Кроссплатформенность
2. Поддержка распределенной кэш-памяти.

3. Большое количество имеющихся решений для распределенных вычислений.
4. Автоматический поиск узлов в локальной сети, что позволяет быстро и удобно масштабировать кластер.

На основе выявленных преимуществ, платформой для дальнейшей разработки распределенного решателя задач глобальной оптимизации был выбрана платформа Apache Ignite.

Выводы к первой главе

1. Интервальный алгоритмы — это мощный инструмент в решении задач глобальной оптимизации многомерной функции.

2. Существует ряд программных продуктов, которые реализуют методы поиска глобального оптимума, но им свойственны некоторые недостатки, одним из которых является детерминированная схема выбора ведущего бруса. Система GOptimum в свою очередь, реализует различные стохастические интервальные методы глобальной оптимизации, имеет открытый исходный код, является кроссплатформенной.

3. Большинство интервальных методов глобальной оптимизации, по своей природе, предрасположено к параллельному исполнению. С одной стороны, в системе GOptimum реализуется параллельный алгоритм поиска глобального экстремума для работы в рамках одного вычислительного узла и масштабировать вычисления на группу узлов, в существующем виде мы не можем. С другой стороны, существуют вычислительные платформы, которые предоставляют разнообразные решения, с помощью которых можно создавать глобальные конструкции для распределенных вычислений. В рамках этой главы был проведен обзор этих инструментов и выбран фреймворк Apache Ignite.

4. Актуальной становится задача разработки адаптированной версии решателя задач глобальной оптимизации на распределенной вычислительной системе.

2. Разработка распределенного решателя задач глобальной оптимизации

2.1 Требования к распределенному решателю задач глобальной оптимизации

На основе анализа, проведенного в предыдущей главе показана актуальность разработки программного средства, которое позволяло бы распределено решать задачи глобальной оптимизации. Были рассмотрены существующие инструменты для решения задач глобальной оптимизации и платформы для построения распределённых вычислений. Среди них были выбраны наиболее подходящие для реализации распределенного решателя задач глобальной оптимизации. Это − система GOptimum и платформа Apache Ignite. Основываясь на использовании этого инструмента и технологии, к распределенному решателю задач глобальной оптимизации необходимо предъявить следующие требования.

1. Распределённый решатель задач глобальной оптимизации должен решать задачи доказательной (гарантированной) глобальной оптимизации, когда требуется не только найти оптимум функции, но и дать гарантию того, что найденное решение является действительно глобальным экстремумом.
2. Условия задачи должны передаваться в приложение с помощью заданного пользователем входного файла. Входной файл должен содержать целевую функцию, переменные, заданные в интервальной форме, задавать допустимое количество итераций, точность поиска глобального оптимума, возможность вывода значения найденного глобального оптимума и значений найденных экстремумов функции, и выбирать тип решаемой задачи (задача минимизации или максимизации).

3. Результаты работы распределенного решателя задач должны записываться в заданный выходной файл. Результатом работы является найденный глобальный оптимум целевой функции, области (брусы), на которых достигается оптимум и время поиска глобального оптимума.
4. Распределенный решатель задач должен быть кроссплатформенным (поддерживать запуск узлов на компьютерах с установленными операционными системами Windows, начиная с версии Windows XP, различных дистрибутивов Linux, и системы Mac OS X начиная с версии 10.6) и разработан на языке программирования Java.
5. Для использования максимально возможного объема аппаратных ресурсов физической машины необходима возможность запуска на ней нескольких программных узлов.
6. Распределённый решатель задач глобальной оптимизации должен быть легко масштабируем на неограниченное количество узлов. Число узлов, необходимое для решения задачи должно задаваться во входном файле.
7. Все физические узлы распределенной системы должны быть объединены в сеть пропускной способностью не ниже 100 Мбит/с.

Согласно требованиям, был разработан проект распределенного решателя задач глобальной оптимизации.

2.2 Архитектура системы GOptimum

Рассмотрим подробнее архитектуру системы GOptimum. Работа системы начинается с класса `FileUserInterface`, который находится в пакете `userinterface`. В конструкторе класса происходит объявление переменных класса отвечающих за вывод ответа, тип оптимизации (минимум или максимум), тип алгоритма, с помощью которого будет осуществляться поиск глобального оптимума, критерии остановки работы алгоритма и структуру данных в которой хранятся переменные и их значения (объявление переменных происходит значениями по умолчанию, на случай, когда какие-либо из параметров не указаны во входном файле), (см. листинг 1). Главным методом класса является

`mainFunction(String args[])`. В аргумент метода подается строка содержащая путь, указанный пользователем до входного и выходного файла. В этой функции происходит вызов методов парсинга входного файла, создание целевой функции, исходного бруса, установка области поиска оптимума, а также установка критерия остановки и запуска процедуры поиска глобального оптимума (см. листинг 2).

Листинг 1. Конструктор класса `FileUserInterface`

```
public FileUserInterface() {
    optimumTypeIsSet = false;
    optimumType = "min";//тип оптимизации
    lineCounter = 0;
    algo = new Bisection_SrtL_CBtC_BigEqS();//тип алгоритма поиска
    stopCriterion = new StopCriterion(algo);//критерий остановки
    hashMapArgs = new HashMap<String, String>();
    reportOptVal = true;//вывод найденного оптимума
    reportOptArea = true;//вывод найденных экстремумов
}
```

Классы, реализующие различные алгоритмы поиска глобального оптимума, находятся в пакете `solvers`. Основная функциональность классов реализуется в конструкторах, в которые создают экземпляры классов `WorkList`, `Chooser` и `Splitte`, а также метод `setLogic(workList, chooser, splitter)`, задающий поведение алгоритма в зависимости от созданных объектов (см. листинг 3). Объект класса `WorkList` – это рабочий список брусов. В зависимости от алгоритма он может быть сортированным и не сортированным. `Chooser` представляет собой метод отбора ведущего бруса из рабочего списка. Он может выбирать случайный брус, либо текущий наилучший брус. Экземпляр класса `Splitter` – это метод дробления ведущего бруса. Дробление может происходить одинаково по всем сторонам, по наибольшей или по случайной стороне.

Листинг 2. Функция `mainFunctioun` класса `FileUsetInterface`

```
public void mainFunction(String args[]) throws Throwable {
      try {
            getAllArgs(args);
            parseArgs(); //парсинг аргументов входного файла
            if (fStr == null){ //проверка на наличие целевой функции
                              //во входном файле
                showToUser("!!! Error. ","Target function hasn't
                been specified.");
                return;
                }
      f = FunctionFactory.newFunction(fStr);//создание целевой
                                            //функции
            Box area = getArea();//создание исходного бруса
            algo.setProblem(f, area);//установка области поиска
                                     //оптимума
            algo.setStopCriterion(stopCriterion);//установка
                                                 //криетрия остановки
            algo.solve();// запуск процедуры поиска
            showResults(algo); //вывод результата
            } catch (Throwable e) {
                reportError(e);
            }
```

В методе создания целевой функции `newFunction(fStr)` класса `FunctionFactory` реализуется разбор строковой переменной с целевой функцией, преобразование формы записи функции в обратную польскую запись [34], затем с помощью метода `simplify(func)`, реализованного в классе `Simplifier`, происходит упрощение целевой функции. Далее вычисляются производные первого и второго порядка. После этого метод возвращает упрощенную функцию вместе с ее производными.

Листинг 3. Конструктор класса `Bisection_SrtL_CBtC_BigEqS`

```
public Bisection_SrtL_CBtC_BigEqS() {
      WorkList workList = new SortedWorkList();
      Chooser chooser  = new CurrentBestChooser(workList);
      Splitter splitter = new BiggestSideEquallySplitter();
setLogic(workList, chooser, splitter);
}
```

В методе `getArea()` создается исходный брус с помощью переменных заданных в интервальной форме. Границы интервалов являются ребрами исходного бруса.

Вызов метода `algo.setProblem(f, area)`, реализованного в классе `BaseAlgorithm`, устанавливает для выбранного алгоритма брус, созданный на предыдущем шаге, добавляя его в рабочий список, а также добавляет в алгоритм целевую функцию созданную и преобразованную ранее.

Вызов метода `algo.setStopCriterion(stopCriterion)`, реализованного в классе `BaseAlgorithm`, задает критерии остановки, которые были установлены по умолчанию, или же заданы пользователем во входном файле.

Вызов метод `algo.solve()`, реализованного в классе `BaseAlgorithm`, запускает процедуру поиска глобального оптимума. Процедура поиска выполняется в цикле, пока не достигнут критерия остановки. На каждой итерации поиска происходит следующие действия алгоритма: при помощи функции `extractNext()`, реализованной в классе `WorkList`, из рабочего списка извлекается брус, выбранный заданным методом выборки `Chooser`, затем брус дробится заданным методом дробления `Splitter`, далее метод `calculateIntervalExtensions(newBoxes)` производит вычисления интервальных расширений на брусьях потомках, после этого все брусы потомки добавляются в рабочий список и затем, происходит проверка на достижение критерия остановки.

Метод `showResults(algo)` записывает результаты работы в выходной файл. После этого работа приложения останавливается. Так работает последовательный алгоритм, реализованный в GOptimum.

Для параллельного запуска поиска глобального оптимума необходимо изменить алгоритм, который мы создаем в конструкторе класса `FileUserInterface` (см. листинг 4).

Листинг 4. Конструктор класса `FileUserInterface` для параллельного алгоритма

```
        public FileUserInterface() {
              optimumTypeIsSet = false;
              optimumType = "min"; //
              lineCounter = 0;
              //тип алгоритма поиска
              baseAlg = new PointIntervalBis_SrtL_CBtC_BigEqS();
              //создаем параллельный алгоритм
              algo = new ParallelExecutor(2, baseAlg);
              stopCriterion = new StopCriterion(algo);
              hashMapArgs = new HashMap<String, String>();
              reportOptVal = true;
              reportOptArea = true;
    }
```

При создании параллельного алгоритма мы создаем объект класса `ParallelExecutor` с двумя аргументами: первый – это количество потоков, с помощью которых мы хотим распараллелить вычисления и второй – это базовый алгоритм, с помощью которого будем искать оптимум. Второй аргумент может быть списком алгоритмов, в таком случае на каждом потоке будет работать разные алгоритмы.

В конструкторе класса `ParallelExecutor` создается массив алгоритмов, размер которых равен числу, заданному в аргументе конструктора, и для каждого алгоритма из массива создается отдельный поток. Далее в конструкторе создается объект класса `AlgorithmsCommunicator`, который выступает

посредником между параллельно работающими алгоритмами. По своей сути AlgorithmsCommunicator это – поток, который работает циклически и забирает рекорды у вычислительных алгоритмов.

Работа методов создания целевой функции, создания исходного бруса и задания критериев остановки при работе параллельного алгоритма не отличаются от последовательного. Однако, метод setProblem(f, area) в параллельной работе дробит исходный брус на число подбрусьев, равное количеству используемых потоков. Затем для каждого алгоритма из массива задается целевая функция и свой начальный подбрус.

По сравнению с последовательной версией изменена работа метода solve(). Теперь используется реализации класса ParallelExecutor. Этот метод запускает работу потоков с алгоритмами и потока AlgorithmsCommunicator. После запуска алгоритмы начинают поиск глобального оптимума на своих подбрусах, а поток AlgorithmsCommunicator поочередно опрашивает статус, который возвращают алгоритмы. Всего четыре типа статусов это RUNNING, EXTERNAL_INTERRUPTED, STOP_CRITERION_SATISFIED, EMPTY_WORKLIST. Они описаны в перечислении OptimizationStatus, которое находится в пакете algorithm. Если алгоритм возвращает статус RUNNING, то это значит, что алгоритм успешно завершил итерацию поиска и у него имеется текущее значение глобального экстремума. Тогда AlgorithmsCommunicator сравнивает это значение с общим рекордом для всех алгоритмов. Если текущее значение больше (в случае задачи минимизации), то AlgorithmsCommunicator обновляет рекорд алгоритма, для последующей отбраковки брусов алгоритма, если же значение меньше, то AlgorithmsCommunicator обновляет рекордное значение для всех алгоритмов. Каждый алгоритм работает до тех пор, пока он не достиг критерия остановки. Если это случилось, то он возвращает статус STOP_CRITERION_SATISFIED и AlgorithmsCommunicator сохраняет значения найденного оптимума. После остановки следующего алгоритма AlgorithmsCommunicator сравнивает границы найденного глобального

Листинг 5. Метод `communicateAlgorithms()` класса `AlgorithmsCommunicator`

```
private void communicateAlgorithms() {
  int failCounter = 0;
  while (failCounter < algorithms.length+1) {
    //задержка для рассинхронизации потоков
    sleep();
    for (int i = 0; i < algorithms.length; i++) {
      OptimizationStatus state = algorithms[i].getState();
      if (state != RUNNING ) {
        switch (state) {
          case EXTERNAL_INTERRUPTED:
           continue;
          case STOP_CRITERION_SATISFIED:
            //сохранение найденного оптимума
            saveFoundOptimumAndArea(algorithms[i]);
            //очистка рабочего списка
            algorithms[i].dropWorkList();
          case EMPTY_WORKLIST:
            //разделение рабочего списка между алгоритмами
            if (!getWorkFromNeighborForThisAlgorithm(i)) {
              failCounter++;
            }
            break;
                default:
             throw new RuntimeException("Unknown OptimizationStatus " +
                                        state);
        }
      }
      else { // RUNNING
        //обновление рекордов
        updateScreeningValueForAlgorithm(i);
        failCounter = 0;
      }
     }
  }
}
```

экстремума с предыдущим и сохраняет наилучшие. Когда прекращается работа всех алгоритмов наилучшее найденное значение возвращается в качестве результата.

После получения очередных границ глобального оптимума и исполнения процедур отбраковки может оказаться, что рабочий список у какого-то алгоритма пуст. Это означает доказательство того, что на подобласти, доставшейся этому потоку, глобального экстремума быть не может. В этом случае алгоритм возвращает статус `EMPTY_WORKLIST`, после чего `AlgorithmsCommunicator`, прочитав этот статус, смотрит, имеются ли рабочие потоки. Если таковые имеются, то он останавливает работу случайного алгоритма и делит его рабочий список пополам, затем алгоритм продолжает работу с одной частью списка, а вторая часть достается алгоритму, отправившему статус `EMPTY_WORKLIST`.

Обработка статусов алгоритмов происходит в методе `communicateAlgorithms()` (см. листинг 5) класса `AlgorithmsCommunicator`. В нем вызываются методы: `saveFoundOptimumAndArea(algorithms[i])`, который сохраняет найденные значения оптимума закончивших работу алгоритмов, `getWorkFromNeighborForThisAlgorithm(i)`, который делит рабочий список и отдает половину простаивающему алгоритму, `updateScreeningValueForAlgorithm(i)`, который сравнивает текущие значения алгоритма с общим рекордом и обновляет рекордные значения (см. листинг 6).

Параллельный многопоточный алгоритм поиска глобальной оптимизации, реализованный в системе GOptimum, понятен и удобен для использования в дальнейших разработках. Учитывая это, именно он взят за основу реализации распределенного решателя задач глобальной оптимизации.

2.3 Платформа Apache Ignite

Фреймворк Apache Ignite предоставляет инструментарий для реализации взаимодействия вычислительных узлов, запущенных на физических машинах. На одной физической машине может быть запущенно несколько вычислительных узлов. Apache Ignite распространяется в архиве. Для установки вычислительного узла достаточно разархивировать скачанный архив. Чтобы узел успешно запустился, необходимо создать переменную окружения IGNITE_HOME и указать путь до папки с разархивированными файлами, также для работы необходима переменная окружения JAVA_HOME с директорией до установленного JDK. После этого нужно запустить пакетный файл ignite.bat, который находится в папке с приложением в директории apache-ignite\bin (для системы Linux и Mac OS необходимо запустить файл ignite.sh). При успешном запуске на экране появится консоль с запущенным приложением (см. рис 2.1).

Рис.2.1. Запущенное приложение Apache Ignite

Таким образом, мы запустили полноценный вычислительный узел. В окне с приложением можно увидеть, что оно запущено в тихом режиме, т.е. протоколы работы не выводятся на экран, а записываются в файл. Далее в окне идет описание операционной системы, платформы java, конфигурации плагинов.

Есть рекомендации по улучшению производительности системы. В самом низу консоли есть надпись, что узел стартовал, отображается его идентификатор и вся информация о топологии сети с количеством запущенных серверов, клиентов, количеством процессоров и доступной памяти.

Настройки работы вычислительного узла хранятся в конфигурационном файле. Примеры некоторых конфигурационных файлов лежат в директории `apache-ignite\examples\config`. Если необходимо запустить узел с конфигурационным файлом, заданным не по умолчанию, нужно с помощью командной строки перейти в директорию apache-ignite\bin и написать, например,

```
ignite.bat examples\config\example-default.xml.
(для Linux ignite.sh examples/config/exmaple-default.xml).
```

При добавлении нескольких узлов в распределенную систему необходимо убедиться, что они имеют одинаковые конфигурационные файлы. Каким образом будет создаваться топология из узлов, определяется в конфигурационном файле. Возможны два случая. В первом узлы находят друг друга автоматически, при условии, что они находятся в одной локальной сети. Это делается с помощью класса `TcpDiscoveryMulticastIpFinder` (см листинг 6).

Во втором случае необходимо самостоятельно написать в конфигурационный файл ip-адреса вычислительных машин, на которых будут запускаться узлы. Для создания топологии в локальной сети с помощью написанных ip-адресов необходимо использовать класс `TcpDiscoveryVmIpFinder`. Необходимо отметить, что на одной физической машине можно запустить несколько вычислительных узлов Apache Ignite. Это актуально при разработке распределенной системы, когда нет достаточного количества физических машин.

После запуска нескольких узлов, информация о топологии сети в консоли изменится (см рис. 2.2). Теперь можно использовать все доступные узлы для решения практических задач.

Листинг 6. Пример сетевых настроек в конфигурационном файле

```
 <property name="discoverySpi">
   <!-- класс реализующий методы для обнаружения узла-->
   <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
     <property name="ipFinder">
    <!-- класс реализующий автоматический метод поиска узлов-->
      <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.
      multicast.TcpDiscoveryMulticastIpFinder">
         <property name="addresses">
           <list>
             <value>127.0.0.1:47500..47509</value>
           </list>
         </property>
       </bean>
     </property>
   </bean>
 </property>
```

Листинг 7. Второй пример сетевых настроек в конфигурационном файле

```
<property name="discoverySpi">
  <!-- класс реализующий методы для обнаружения узла-->
  <bean class="org.apache.ignite.spi.discovery.tcp.TcpDiscoverySpi">
    <property name="ipFinder">
<!-- класс реализующий метод поиска по заранее заданным ip адресам-->
      <bean class="org.apache.ignite.spi.discovery.tcp.ipfinder.
                                    vm.TcpDiscoveryVmIpFinder">
        <property name="addresses">
          <list>
            <value>192.168.1.10</value>
            <value>192.168.1.11</value>
            </list>
        </property>
      </bean>
    </property>
  </bean>
</property>
```

Узлы кластера могут работать в двух режимах – клиент и сервер. В режиме сервера узлы участвуют в вычислениях, кэшировании, потоковой обработке данных, транзакциях и т.п., в то время как клиентские узлы создают распределенные кэши, задания, которые выполняются на узлах-серверах. По умолчанию узлы-клиенты не участвуют в различные рода вычислениях. Настройка режима запуска узла происходит в конфигурационном файле, по умолчанию запускаются узлы-серверы. Необходимо отметить, что все узлы распределенной системы имеют равноправное значение

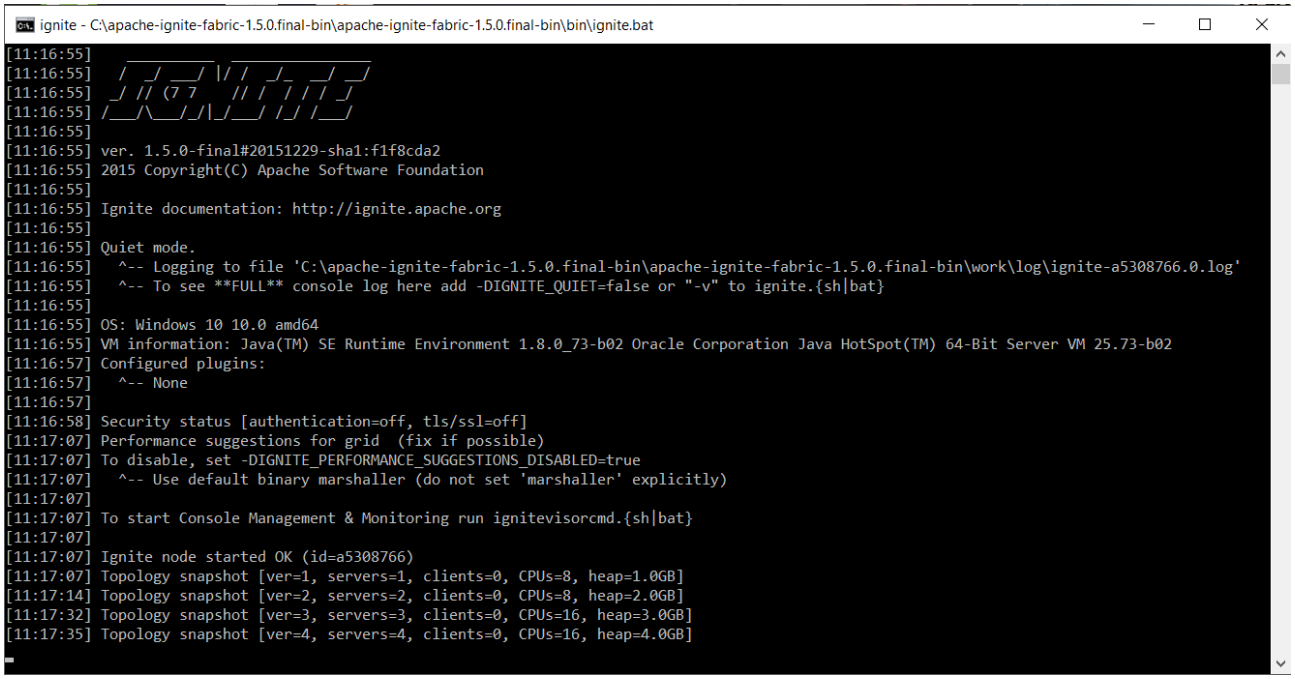

Рис.2.2. Запущенное приложение Apache Ignite

Для того чтобы начать работать с Apache Ignite в проекте можно использовать фреймворк для автоматизации сборки проектов Maven [28]. Для этого в файле pom.xml проекта необходимо написать зависимости Apache Ignite (см. листинг 8). После загрузки все репозиториев можно начинать пользоваться библиотеками платформы.

В проекте вычислительный узел запускается с помощью строки кода

```
Ignite ignite=Ignition.start("examples/config/exampleIgnite.xml"),
```

где `Ignite` – это интерфейс для всего Ignite API, `Ignition` – это класс, реализующий методы запуска, остановки, просмотра состояния, перезапуска, изменения режима работы узла. Метод `start("examples/config/exampleIgnite.xml")` запускает вычислительный узел, в аргумент метода подается конфигурационный файл.

Листинг 8. Зависимости Apache Ignite для maven

```
<dependency>
    <groupId>org.apache.ignite</groupId>
    <!-- базовый API -->
    <artifactId>ignite-core</artifactId>
    <version>${ignite.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <!-- конфигурации xml для spring -->
    <artifactId>ignite-spring</artifactId>
    <version>${ignite.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.ignite</groupId>
    <!-- конфигурацмя xml для SQL запрос -->
    <artifactId>ignite-indexing</artifactId>
    <version>${ignite.version}</version>
</dependency>
```

Далее рассмотрим, какие инструменты предоставляет платформа Apache Ignite для реализации распределенных приложений.

Узлы распределённой системы можно объединить в кластер или несколько кластеров. Функциональность кластера обеспечивает интерфейс `IgniteCluster`. С помощью этого интерфейса можно запускать или останавливать удаленные узлы кластера, получать список всех членов кластера, а также создавать логические группы из узлов кластера. Последнее делается при помощи интерфейса `ClusterGroup`. С помощью него можно группировать узлы

кластера для различных целей, например, в случае, когда необходимо запустить вычисления только на удаленных узлах, или производить обмен сообщениями только между конкретными группами.

Для запуска вычислений на узлах и группах кластера не нужно вручную разворачивать Java код на каждом узле распределенной системы, а также повторно разворачивать его после каких-либо изменений. Это возможно благодаря технологии `peer class loading`, с помощью которой узлы обмениваются байт-кодом приложения. Это происходит только один раз, во время запуска вычислений на узлах.

Apache Ignite поддерживает распределенную передачу сообщений между узлами или между группами кластера. Имеется поддержка широковещательного отправления всем узлам кластера. Передача сообщений осуществляется с помощью интерфейса `IgniteMessaging` (см листинг 9). Сообщения могут быть оправлены в упорядоченной или неупорядоченной манере. Если необходимо чтобы узлы получили сообщения в том порядке, в котором они были отправлены, то используется метод `sendOrdered()`, если же это не нужно, то используется метод `send()`. В качестве аргумента методов необходимо использовать адресат сообщения (это может быть группа кластера или идентификатор узла) и сообщение в виде переменной или какой-либо структуры данных. Метод `localListen()` используется для приема сообщений на конкретном узле, метод `remoteListen()` для приема сообщения на нескольких узлах или группе кластера (см. листинг 10).

Листинг 9. Примеры создания сообщений

```
Ignite ignite = Ignition.ignite();
// пример сообщения для всего кластера
IgniteMessaging msg = ignite.message();
// пример сообщения для группы кластера состоящей из удаленных узлов
IgniteMessaging rmtMsg = ignite.message(ignite.cluster().forRemotes());
```

Листинг 10. Прием и отправка сообщений

```
Ignite ignite = Ignition.ignite();
// создание сообщения для удаленной группы узлов
IgniteMessaging rmtMsg = ignite.message(ignite.cluster().forRemotes());
// добавление «слушателя» для приема сообщений удаленными узлами
// в качестве аргумента передается тема сообщения и структура с
// идентификатором узла отправившим сообщение и самим сообщением
rmtMsg.remoteListen("MyOrderedTopic", (nodeId, msg) -> {
    System.out.println("Received ordered message [msg=" + msg + ", from="
    + nodeId + ']');
    return true; // возвращает true по завершению приема сообщения
});
// отправка сообщений удаленным узлам
for (int i = 0; i < 10; i++)
    rmtMsg.sendOrdered("MyOrderedTopic", Integer.toString(i));
```

Apache Ignite реализует технологии для создания и использования распределенного кэша. Интерфейс `IgniteCache` предоставляет методы для хранения и извлечения данных, а также выполнения различных запросов из кэша. Кэш может работать в трех режимах.

1. Разделенный режим. Этот режим является самым масштабируемым. При создании кэша с разделенным режимом набор данных делится на разделы, все разделы распределены поровну между участвующими узлами, по существу создавая огромное хранилище для кэшированных данных в распределенной памяти.
2. Реплицированный режим. В этом режиме все данные реплицируются на каждом узле кластера. Такой режим обеспечивает максимальную доступность данных на каждом узле, однако, в таком случае каждое обновление данных должно распространяться на все остальные узлы, что может повлиять на производительность и масштабируемость. Размер реплицированного кэша ограничен объемом доступной памяти на с наименьшим количеством оперативной памяти.

3. Локальный режим. Это наиболее легкий, с точки зрения производительности, режим работы кэш-памяти, так как данные не распространятся на все узлы распределенной системы, а находятся локально. Такой режим идеально подходит для сценариев, где происходит только чтение данных, либо периодическое обновление с некоторой частотой истечения срока годности.

Создание кэша показано в листинге 11.

Листинг 11. Создание распределенного кэша

```
Ignite ignite = Ignition.ignite();
// создание кэша с парой ключ значение, Integer String и
// именем newCache
IgniteCache<Integer, String> cache = ignite.cache("newCache");
```

Apache Ignite с помощью интерфейса `IgniteCompute` предоставляет методы для запуска многих типов вычислений на узлах или группах кластера. Вычисления будут гарантированно выполняться, пока существует хотя бы один рабочий узел. В случае сбоя, система балансировки нагрузки выбирает следующий доступный узел для выполнения задачи. На листинге 12 приведен пример создания распределенного вычисления для удаленной группы узлов.

Листинг 12. Создание вычисления для группы кластера

```
Ignite ignite = Ignition.ignite();
// создание группы кластера
ClusterGroup remoteGroup = ignite.cluster().forRemotes();
// создание вычисления для группы кластера remoteGroup
IgniteCompute compute = ignite.compute(remoteGroup);
```

Apache Ignite в вычислительной системе позволяет запускать распределенные вычисления внутри кластера, в том числе использовать методы `runnable` и `callable` из Java. Имеется поддержка широковещательного и асинхронного запуска вычислений. Для запуска широковещательного задания используется

функция `broadcast()`, в аргумент которой подается процедура, которую необходимо выполнить на узлах группы кластера. Для запуска асинхронных заданий на узлах кластера необходимо при создании вычисления включить режим асинхронности `withAsync()`.

В Apache Ignite существует реализация парадигмы MapReduce. Для этого используется API `ComputeTask`. `ComputeTask` определяет задания для выполнения на кластере и определяет, какие узлы будут выполнять эти задания. Он также определяет, как обрабатывать результаты выполнения заданий. В разрабатываемом приложении должны быть реализованы методы интерфейса `ComputeTask` – `map()` и `reduce()`. Метод `map()` создаёт задания и затем запускает их на рабочих узлах. Метод `reduce()` вызывается, когда все задания выполнены. Метод получает список всех результатов и возвращает конечный результат вычислений. Подробнее со всеми инструментами Apache Ignite можно ознакомиться в документации [29].

2.4 Распределенный решатель задач глобальной оптимизации

В соответствии с требованиями, изложенными в пункте 2.1, с помощью инструмента для поиска глобального оптимума GOptimum, описанного в пункте 2.2, и платформы для построения распределенных вычислений Apache Ignite, описанной в пункте 2.3, разработано программное обеспечение «Распределенный решатель задач глобальной оптимизации».

Распределенный решатель задач глобальной оптимизации построен по схеме “master-slave”, где имеется один главный узел и несколько подчиненных (рабочих) узлов (см. рис 2.3). Главный узел отвечает за подготовку данных для решения задачи, создание групп кластера, создание распределенного кэша, создание и запуск заданий с процедурами поиска экстремума на рабочих узлах, прием сообщений и реагирование на них, обработка результатов поиска глобального оптимума и вывод результата. На рабочих узлах запускаются

процедуры поиска глобального экстремума на заданном брусе, взаимодействие с главным узлом посредством сообщений, добавление результатов поиска для обработки на главном узле.

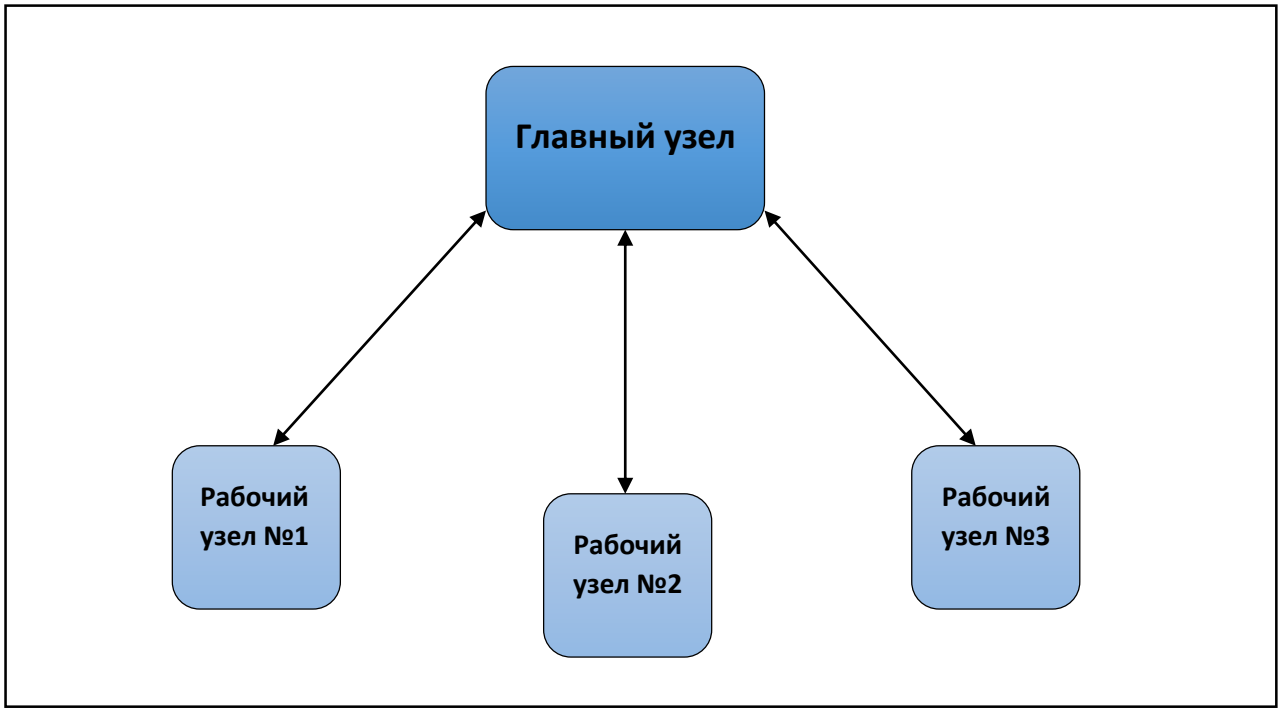

Рис.2.3. Схема взаимодействия узлов распределенного решателя задач глобальной оптимизации

Кратко схема работы распределенного решателя задач глобальной оптимизации представлена на рис 2.4.

Подробная схема работы распределенного решателя задач глобальной оптимизации включает следующие шаги:

1. Настройка входного файла. Входной файл с описанием задачи практически не отличается от входного файла системы GOptimum, за исключением добавления обязательного параметра `NumNodes`, с помощью которого задается количество узлов, на которых будет вычисляться глобальный оптимум.

2. Запуск необходимого для решения задачи оптимизации количества рабочих узлов на физических машинах, объединенных в локальную сеть. После запуска узлы объединятся в вычислительную сеть.
3. Запуск главного узла. В качестве параметров запуска главного узла выступает путь до входного и выходного файла. Первым делом на главном узле происходит парсинг входного файла. После получения всех необходимых данных, создается целевая функция, целевой брус и массив алгоритмов поиска глобального оптимума, каждый из которых будет запущен на рабочих узлах. Алгоритм поиска может различаться на каждом узле. Затем исходный брус дробится на подбрусы и каждый из них последовательно передается созданным на предыдущем этапе алгоритмам. Количество подбрусов равно количеству используемых рабочих узлов. Далее создается две кластерные группы, первая включает в себя рабочие узлы, вторая включает главный узел. После этого на рабочем узле создается четыре распределенных кэша:
   - `currentGlobalOptimum` – используется для обновления текущего наилучшего значения оптимума для всех рабочих узлов кластера;
   - `result` – используется для записи результирующего списка оптимумов областей на которых достигается оптимальное значение;
   - `jobForNodeWithEmptyWorklist` – используется для передачи части рабочего списка, в случае, когда необходимо разделить рабочий список между двумя узлами;
   - `indexesNodesWithEmptyWorklist` – используется для записи идентификатора алгоритма, который имеет пустой рабочий список и ожидает получения новой порции брусов от другого алгоритма. Далее происходит запуск слушателей сообщений от рабочих узлов.
4. На главном узле происходит создание списка исполняемых процедур для рабочих узлов. Каждая процедура включает в себя алгоритм поиска глобального оптимума из массива алгоритмов, который мы создали на предыдущем этапе.

5. Главный узел одновременно запускает исполняемые процедуры на рабочих узлах. После запуска каждый рабочий узел отправляет сообщение на главный узел со своим идентификатором и идентификатором алгоритма, который на нем запущен. Главный узел сохраняет эти данные в структуре данных `hashmap`. Затем рабочие узлы переходят непосредственно к алгоритму поиска глобального экстремума.
6. Во время выполнения алгоритма, после каждой итерации поиска, узел сравнивает свой результат с общим, наилучшим результатом для всех узлов, который записывается в `currentGlobalOptimum`. Если его результат лучше он обновляет значение общего результата в кэше, если хуже, то он использует общий результат для отбраковки брусов в своем рабочем списке и затем продолжает поиск.

   Если после процедуры отбраковки окажется так что на всем его рабочем списке границы оптимума хуже, чем общий результат, то узел отправит сообщение `NEED_JOBS` на главный узел. Главный узел, приняв это сообщение, добавляет идентификатор алгоритма с пустым рабочим списком в `indexNodesWithEmptyWorklist`. Работающие узлы после заданного количества итераций проверяют, имеется ли простаивающий узел, если такой имеется, то первый увидевший это узел делится половиной своего рабочего списка, добавляя его в `jobForNodeWithEmptyWorklist`, и затем, убирает значение идентификатора из `indexNodesWithEmptyWorklist`, после чего продолжает работу с оставшейся половиной. Рабочий список из `jobForNodeWithEmptyWorklist` отдается для работы простаивающему узлу.
7. Остановка работы рабочего узла происходит, если достигнут критерий остановки алгоритма поиска глобального экстремума. После остановки алгоритма рабочие узлы записывают найденный глобальный оптимум в кэш `result` и отправляют на главный узел сообщение

`Stop_Criterion_Satisfied`, после приема сообщения главный узел сохраняет идентификатор узла, отправившего сообщение.

8. После остановки всех рабочих узлов главный узел забирает результаты из распределенного кэша и сравнивает их. Наилучший результат он выводит на экран, а также записывает в выходной файл. После этого главный узел прекращает работу.

Рис.2.4. Схема работы распределенного решателя задач

Рассмотрим подробнее алгоритм работы распределённого решателя задач глобальной оптимизации. Запускает алгоритм функция `mainFunction(String[] args)` класса `FileUserInterface`. Первое действие алгоритма – это запуск главного узла с конфигурационным файлом `masterNodeConfig.xml`, следующей строкой кода:

```
Ignite ignite = Ignition.start("gridConfig/masterNodeConfig.xml").
```

Конфигурационный файл выглядит следующим образом. Сначала активируется технология развертывания приложения `peer class loading`, затем выбирается режим развертывания `shared`, при котором при запуске приложения на рабочие узлы будет автоматически загружаться байт-код необходимых классов приложения (см. листинг 13). Далее настраиваются четыре распределенных кэша. Устанавливаются имена, режимы работы, начальные размеры и отключается режим бэкапа данных (см. листинг 14).

Листинг 13. Настройка развертывания приложения

```
<!--активация peer class loading -->
<property name="peerClassLoadingEnabled" value="true"/>
<!--выбор режима развертывания -->
<property name="deploymentMode" value="SHARED"/>
```

После запуска главного узла запускается процедура парсинга входного файла. Для установки количества узлов, на которых будет вычислять глобальный оптимум была реализован метод `setNumNodes(value)`, который принимает количество узлов из файла и записывает в переменную `numberNodes`. Остальные методы парсинга входного файла не отличаются от реализованных в GOptimum.

После парсинга происходит создание массива алгоритмов, с помощью которых будем искать глобальный оптимум. Для этого используется конструктор класса `GridParallezation`. В конструктор подается количество узлов и заданный алгоритм. Также в конструкторе класса `GridParallezation`

создается структура hashmap, в которой будут храниться ключ-значения идентификатор узла и идентификатор алгоритма.

Листинг 14. Пример настройки распределенного кэша

```
<property name="cacheConfiguration">
  <list>
    <bean class="org.apache.ignite.configuration.CacheConfiguration">
         <!--установка имени кэша -->
         <property name="name" value="currentGlobalOptimum"/>
         <!--установка режима работы кэша-->
         <property name="cacheMode" value="REPLIACATED"/>
         <!-- настройка стартового размера-->
         <property name="startSize" value="#{100 * 100 * 100}"/>
         <!-- отключение режима бэкапа данных-->
         <property name="backups" value="0"/>
    </bean>
  </list>
</property>
```

Метод создания целевой функции newFunction() класса FunctionFactory, метод создания бруса области определения getArea() класса FileUserInterface не отличаются от реализованных в GOptimum.

Метод установки критериев остановки setStopCriterion() реализован в классе GridParallelization, его задача состоит в установке критериев остановки для каждого алгоритма поиска глобального оптимума из массива .

Метод setProblem(), реализованный в классе GridParallelization, дробит исходный брус на подбрусы. Затем последовательно устанавливает целевую функцию и каждый подбрус для алгоритмов поиска глобального оптимума из массива. Аргументами метода служат целевая функция и исходный брус.

Метод solve(ignite) реализован в классе GridParallelization. В качестве аргумента метода передается объект класса Ignite. Работа алгоритма

метода проходит в несколько этапов. Этап первый − это создание кластера `clusterFindOpt` с помощью строки кода:

```
IgniteCluster clusterFindOpt = ignite.cluster(),
```

создание группы кластера worksClusterGroup, состоящей из рабочих узлов:

```
ClusterGroup workClusterGroup = clusterFindOpt.forRemotes(),
```

создание группы кластера master, состоящей из рабочего узла:

```
ClusterGroup masterClusterGroup = clusterFindOpt.forNode(master),
```

где master – это главный узел. Затем происходит создание кэша для обновления наилучшего, текущего, общего значения глобального оптимума для все узлов – `currentGlobalOptimum`, кэша для сохранения результатов работы узлов – `result`, кэша для обмена рабочим списком – `jobForNodeWithEmptyWorklist`, и кэша для хранения индексов простаивающих алгоритмов `indexNodesWithEmptyWorklist`. Общее текущее значение глобального оптимума для всех узлов будет храниться в кэше `currentGlobalOptimum` в ячейке с нулевым индексом. Для сравнения общего значения со значением алгоритма, который первый закончит первую итерацию поиска в `currentGlobalOptimum` добавляется `Double.POSITIVE_INFINITY` в ячейку с индексом 0. После сравнения он заменит значение бесконечности найденным оптимумом. Все остальные узлы будут сравнивать найденные оптимумы с новым значением и, если необходимо, изменять его. На листинге 15 приведен пример создания `currentGlobalOptimum`.

Листинг 15. Пример создания `currentGlobalOptimum`

```
//создание currentGlobalOptimum
IgniteCache<Integer,Double> currentGlobalOptimum = ignite.cache();
//добавление первого значения в currentGlobalOptimum
currentGlobalOptimum.put(0, Double.POSITIVE_INFINITY);
```

Второй этап – это создание “слушателей” сообщений от рабочих узлов (см. листинг 16). Слушатели различаются по теме сообщения, которое отсылает

рабочий узел. Темы принимаемых сообщений на главном узле могут быть: “ALG_NUM”, “NEED_JOBS”, “STOP_CRITERION_SATISFIED”, “RESTART”. Позже рассмотрим принимаемые сообщения более детально.

Листинг 16. Запуск слушателя сообщений с темой ALG_NUM

```
ignite.message().localListen(ALG_NUM,
                             new IgniteBiPredicate<UUID, Integer>()
  {
    @Override public boolean apply(UUID nodeId, Integer msgString){
    ...
    // далее происходит разбор сообщения и реакция главного
    // узла в зависимости от темы ссообщения
    }
  }
);
```

Третий этап работы алгоритма метода `solve(ignite)` – это запуск метода

```
createAndStartComputeInNode (worksClusterGroup,
                             masterClusterGroup,
                             currentGlobalOptimum,
                             indexNodesWithEmptyWorklist,
                             result,ignite).
```

В качестве аргументов подаются созданные группы кластера, кэши и объект `ignite`. В методе происходит создание объекта compute, с помощью которого будет запускаться вычисления на узлах группы кластера `worksClusterGroup`. Затем происходит создание списка исполняемых процедур `runs` и добавление в него подготовленных алгоритмов поиска глобального оптимума, которые будет выполняться на рабочих узлах кластера. После этого исполняемые процедуры запускаются на узлах кластера.

После запуска исполняемых процедур первым рабочие узлы пересылают сообщение с темой “ALG_NUM” на главный узел. Тело этого сообщения — это переменная содержащая идентификатор алгоритма, запущенного на узле

отправителе. Также сообщение содержит идентификатор самого узла. Главный узел, приняв это сообщение, добавляет в `hashmap algorithmForNode` значения идентификаторов. После приема всех сообщений содержимое `algoithmForNode` отображается в окне консоли главного узла (см. листинг 17).

Метод
```
startAlgorythm(ignite, masterClusterGroup,
                       currentGlobalOptimum,
                       result,
                       indexNodesWithEmptyWorklist,
                       jobForNodeWithEmptyWorklist)
```
реализован в классе `AlgorithmForGrid`. В качестве аргументов используется объект `ignite`, группа кластера с главным узлом, и четыре кэша. Первым делом в методе создаётся слушатель сообщений для рабочих узлов. Принимаемые сообщения на рабочих узлах могут быть с темой "`GET_NEW_WL`".

После запуска слушателей сообщений в методе `startAlgorythm` вызывается метод

```
solve(ignite, masterClusterGroup,
                      currentGlobalOptimum,
                      result,
                      indexNodesWithEmptyWorklist,
                      jobForNodeWithEmptyWorklist).
```

Рассмотрим его работу подробнее. Сначала создается переменная `tStat` которая содержит статус работы алгоритма поиска после каждой итерации. Значение статуса возвращает функция iterate(currentGlobalOptimum), которая на каждой итерации считает значения на выбранном брусе, проводит процедуру отбраковки брусов в рабочем списке и проверяет достижение критериев остановки. В качестве аргумента передается кэш со значением текущего глобального оптимума, с помощью которого происходит процедура отбраковки. Если рабочий список после отбраковки становится пустым, метод возвращает статус "`EMPTY_WORKLIST`", если достигнут критерий остановки, возвращает статус "`STOP_CRITERION_SATISFIED`" (см. листинг 18).

Листинг 17. Метод createAndStartComputeInNode

```
public void createAndStartComputeInNode (ClusterGroup worksClusterGroup,
            ClusterGroup masterClusterGroup,
            IgniteCache<Integer, Double> currentGlobalOptimum,
            IgniteCache<Integer, Integer> indexNodesWithEmptyWorklist,
            IgniteCache<Integer, Result> result,
            Ignite ignite){
  //создание «вычисления» для группы кластера workClusterGroup
  IgniteCompute compute = ignite.compute(worksClusterGroup);
  //создание списка исполняемых процедур runs
  Collection<IgniteRunnable> runs = new ArrayList<>(numberNodes);
  //для каждого алгоритма из массива созданных алгоритмов
  for (AlgorithmForGrid alg: algorithms) {{
    //добавление исполняемой процедуры в список runs
    runs.add(new IgniteRunnable() {
      @Override
      //все что будет выполняться на каждом узле
      public void run() {
        //записываем идентификатор алгоритма в переменную AlgNum
        Integer algNum = alg.getId();
        //создаем сообщение для отправки на главный узел
        IgniteMessaging msg = ignite.message(masterClusterGroup);
        //отправляем сообщение с темой "AlgNum"
        //и телом-переменной algNum
        msg.send("AlgNum", algNum);
        //метод startAlgorythm() запускает работу алгоритмов на узлах
        alg.startAlgorythm(ignite, masterClusterGroup,
                                   currentGlobalOptimum,
                                   result,
                                   indexNodesWithEmptyWorklist,
                                   numberNodes,
                                   jobForNodeWithEmptyWorklist);
      }
    });
  }
  //одновременный запуск всех процедур на узлах
  compute.run(runs);
}
```

Листинг 18. Метод `solve`

```
public void solve(Ignite ignite,
     ClusterGroup masterClusterGroup,
     IgniteCache<Integer, Double> currentGlobalOptimum,
     IgniteCache<Integer, Result> result,
     IgniteCache<Integer, Integer> indexNodesWithEmptyWorklist,
     IgniteCache<Integer, WorkList> jobForNodeWithEmptyWorklist){
  //создание статуса
  OptimizationStatus tStat;
  do{
    //каждые 300 и итераций
    if(iteration==k){
      //смотрим имеется ли в кэше indexNodesWithEmptyWorklist
      //индекс узла с пустым рабочим списком
      if((indexNodesWithEmptyWorklist.get(0)!=0)&&
        //смотрим достаточный ли размер рабочего списка для
        //разделения
        (getSizeWorkList()>500)){
          //если все условия выполнены делим рабочий список
          divideWork(ignite, masterClusterGroup,
                             jobForNodeWithEmptyWorklist,
                             indexNodesWithEmptyWorklist,
                             currentGlobalOptimum);
      }
        k=k+300;
    }
    //метод производящий 1 одну итерацию поиска и
    //возвращающий статус работы
    tStat = iterate(currentGlobalOptimum);
    status = tStat;
    if(status != RUNNING){
      switch (status) {
        //если статус EMPTY_WORKLIST
        case EMPTY_WORKLIST:
          IgniteMessaging msg;
          Msg = ignite.message(masterClusterGroup);
          //отправляем сообщение NEED_JOBS на главный узел
          msg.send("NEED_JOBS", getId());
          break;
        //если статус STOP_CRITERION_SATISFIED
        case STOP_CRITERION_SATISFIED:
          //сохраняет результат поиска в кэш result
          saveFoundOptimumAndArea(result,
                                  currentGlobalOptimum)
          //очищает рабочий список
          dropWorkList();
          IgniteMessaging msg2;
          msg2= ignite.message(masterClusterGroup);
          //отправляет сообщение STOP_CRITERION_SATISFIED на
          //главный узел
          msg2.send("STOP_CRITERION_SATISFIED", getId());
          break;
```

Листинг 18 (продолжение). Метод `solve`

```
      }
    }
    else{
      //если статус RUNNING
      updateCurrentValueForAlgorithm(currentGlobalOptimum);
    }
    iteration++;
    //алгоритм работает пока статус RUNNING
  } while (status == RUNNING);
}
```

В зависимости от имеющегося статуса происходят следующие действия. Если метод `iterate(currentGlobalOptimum)` возвращает статус “`RUNNING`” происходит вызов метода `updateCurrentValueForAlgoithm` в который передается кэш `currentGlobalOptimum` в качестве аргумента. В методе происходит сравнение значения общего текущего наилучшего значения оптимума, взятого из кэша `currentGlobalOptimum` со значением оптимума найденного после произведенной итерации конкретным алгоритмом на узле. Метод `getAlgorithmCurrentValue()` возвращает значение оптимума конкретного алгоритма. С помощью метода `currentGlobalOptimum.get(0)` мы из нулевой ячейки достаем текущее наилучшее значение оптимума. Метод `get()` – атомарный, это значит, что доступ к кэшу будет ограничен только одним узлом. После этого взятые значения сравниваются. Если общее значение оптимума больше, чем значения оптимума у алгоритма, то алгоритм обновляет свой рекорд, который использует для отбраковки брусов в своем рабочем списке и затем продолжает поиск. В противном случает алгоритм обновляет общее значение оптимума и также продолжает поиск оптимума.

Если возвращен статус “`EMPTY_WORKLIST`” алгоритм отправляет сообщение с темой “`NEED_JOBS`” и своим идентификатором на главный узел. После этого рабочий узел заканчивает работу и ожидает дальнейших указаний. Главный узел, приняв сообщение добавляет идентификатор алгоритма в кэш `indexNodesWithEmptyWorklist`.

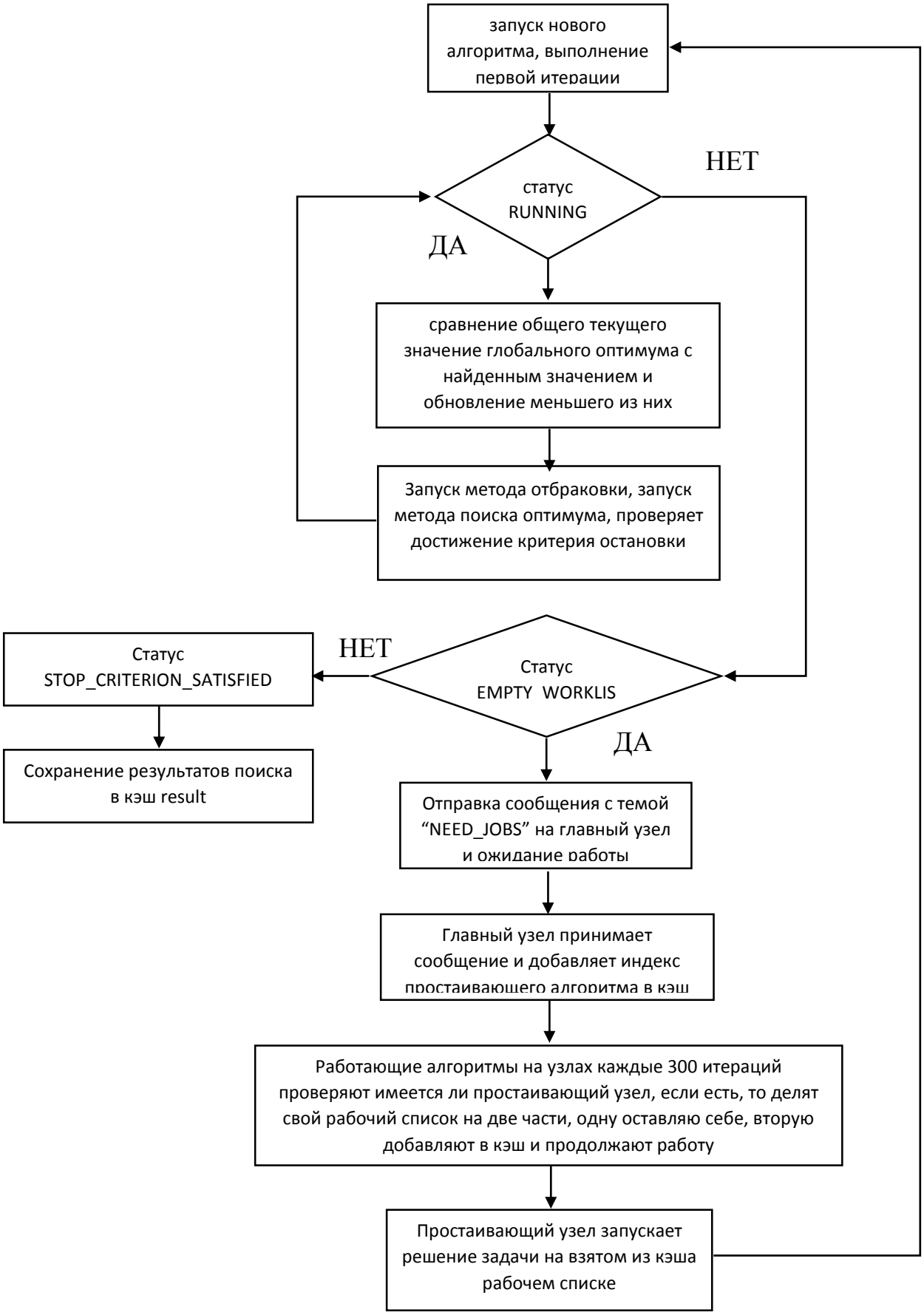

Рис 2.5. Блок-схема работы алгоритма поиска глобального оптимума на узлах распределенной системы

Через заданное количество итераций алгоритмы на узлах проверяют, имеется ли значения в кэше `indexNodesWithEmptyWorklist`. Если таковое имеется, то алгоритм, который первый это обнаружил, проверяет размер своего рабочего списка, если размер удовлетворяет заданному числу то он приостанавливает свою работу и запускает метод

```
divideWork (
      Ignite ignite,
      ClusterGroup masterClusterGroup,
      IgniteCache<Integer, WorkList> jobForNodeWithEmptyWorklist,
      IgniteCache<Integer, Integer> indexesNodesWithEmptyWorklist,
      IgniteCache<Integer, Double>  currentGlobalOptimum).
```

Этот метод делит рабочий список на две половины, одну половину оставляет себе, а вторую добавляет в кэш `jobForNodeWithEmptyWorklist`, далее обнуляет значение в кэше `indexNodesWithEmptyWorklist` и затем отправляет сообщение на главный узел с с темой "`RESTART`". После этого алгоритм продолжает работу на своей половине рабочего списка. Главный узел, приняв сообщение, отправляет сообщение с темой "`GET_NEW_WL`" на простаивающий узел, чтобы тот забрал из кэша `jobForNodeWithEmptyWorklist` рабочий список и запустил алгоритм поиска над ним. Если за время, которое потребовалось на обнаружение простаивающего узла, появился еще один узел с пустым рабочим списком, то индекс его алгоритма, добавляется в стек `idAlgWithEmptyWorklist` на главном узле. После обнуления кэша `indexNodesWithEmptyWorklist` значение забирается из стека и добавляется в кэш `indexNodesWithEmptyWorklist`.

Если возвращен статус алгоритма "`STOP_CRITERION_SATISFIED`", то с помощью метода

```
saveFoundOptimumAndArea(
                IgniteCache<Integer, Result> result,
                IgniteCache<Integer, Double> currentGlobalOptimum)
```

алгоритм добавляет найденное значение оптимума в кэш result. После чего очищает свой рабочий список и отправляет сообщение с темой

"STOP_CRITERION_SATISFIED" на главный узел, чтобы проинформировать его об окончании работы.

Блок-схема алгоритма поиска глобального оптимума на узлах кластера представлена на рис. 2.5.

Четвертый этап работы метода `solve(ignite)` – поиск наилучшего результата среди добавленных в кэш `result`. Это делается с помощью метода `getWinner(result)`. Запуск метода происходит, когда все узлы прекратили свою работу. Метод забирает результаты из кэша `result`, записывает в созданную структуру типа `Result`, затем сравнивает их между собой и записывает в переменную `indexWinner` индекс наилучшего результата поиска глобального оптимума. После этого происходит очистка всех типов кэш памяти. На этом работа метода `solve(ignite)` заканчивается.

С помощью метода `showResult(algo)` реализованного в классе `FileUserInterface` значение глобального оптимума выводится в окне консоли главного узла и записывается в выходной файл. На этом работа распределенного решателя задач глобальной оптимизации заканчивается.

Описанный алгоритм определяет прочие вспомогательные элементы архитектуры, которые не описаны в этой работе. Такой алгоритм работы распределённого решателя задач глобальной оптимизации был реализован в виде приложения работоспособность которого исследована в следующей главе.

Выводы ко второй главе

1. Выработаны функциональные требования, на основании которых спроектировано приложение «Распределенный решатель задач глобальной оптимизации».
2. Рассмотрена архитектура приложения GOptimum и инструменты фреймворка для распределенных вычислений Apache Ignite, необходимые для реализации распределенного решателя задач глобальной оптимизации.
3. Реализован распределенный решатель задач глобальной оптимизации.

3. Тестирование и экспериментальное оценивание эффективности распараллеливания

3.1 Тестирование корректности вычислений глобального оптимума

Реализованный распределенный решатель задач глобальной оптимизации, описанный в пункте 2.4 тестировался на распределенной системе, состоящей из 4 физических узлов.

Узел 1: Процессор - AMD FX - 8120 Eight-Core 3.10 GHz.

Память - 10 Гб DDR3-1333.

Операционная система – Windows 7 x64.

Узел 2: Процессор – Intel Core i7-4720HQ CPU 2.60 GHz.

Память – 8 Гб DDR3-1600.

Операционная система – Windows 10 x64.

Узел 3: Процессор – Intel Core i3-3150 CPU 3.50 GHz.

Память – 8 Гб DDR-1333.

Операционная система – Windows 7 x64.

Узел 4: Процессор – Intel Core i3-2120 CPU 3.30 GHz.

Память – 8 Гб DDR-1333.

Операционная система – Windows 7 x64.

Взаимодействие между физическими узлами осуществлялось с помощью сетевых интерфейсов пропускной способностью 100 Мб/с. На каждом физическом узле был запущен вычислительный узел Apache Ignite с автоматическим поиском узлов внутри сети.

Тестовые функции были взяты из [30,31,32,33].

Для функции Ackley (рис.3.1):

$$f(x,y) = -20\exp\left(-0.2\sqrt{0.5(x^2+y^2)}\right) - \exp\left(0.5(\cos(2\pi x)+\cos(2\pi y))\right) + e + 20,$$

где $-5 \leq x, y \leq 5$, минимум был локализован в $f(0,0) = 0$ за 3.8 сек.

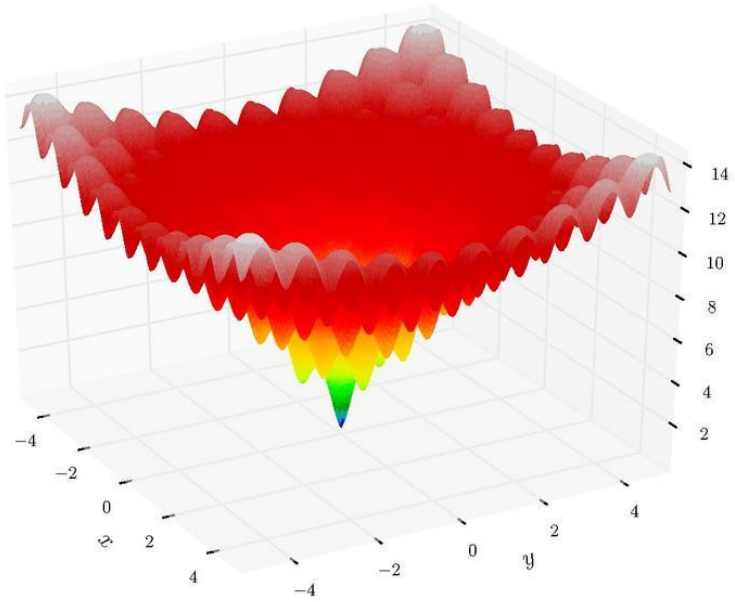

Рис.3.1. Функция Ackley

Для функции Branin с двумя переменными (рис.3.2):

$$f(x) = \left(x_2 - \frac{5.1x_1^2}{4\pi^2} + \frac{5x_1}{\pi} - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos(x_1) + 10,$$

где $-5 \leq x_1 \leq 10, 0 \leq x_2 \leq 15$, минимум был локализован в $f([-\pi, 12.275], [\pi, 2.275], [3\pi, 2.425]) = 0.3978$ за 4.1 сек.

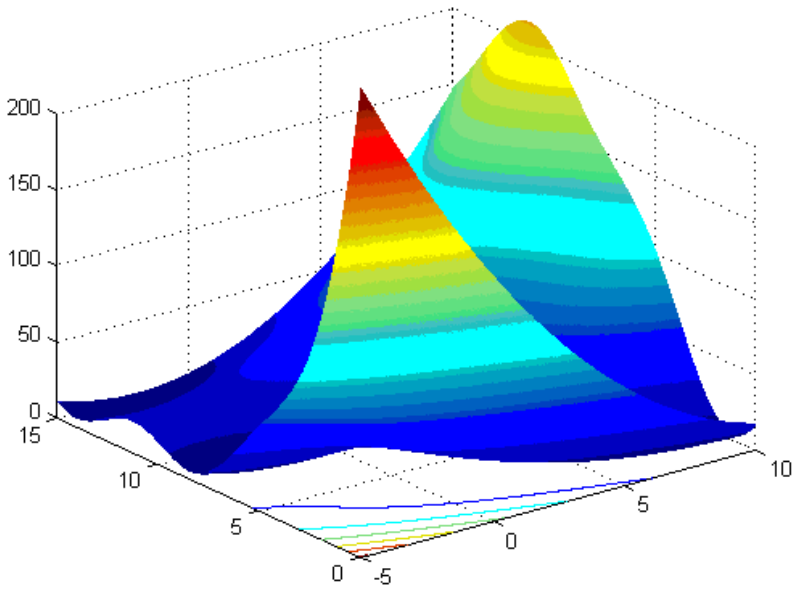

Рис.3.2. Функция Branin

Для функции Beale (рис.3.3):

$$f(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2,$$

где $-4.5 \leq x_i \leq 4.5,\ i = 1,2$, минимум был локализован в $f(3,0.5) = 0$ за 5.3 сек.

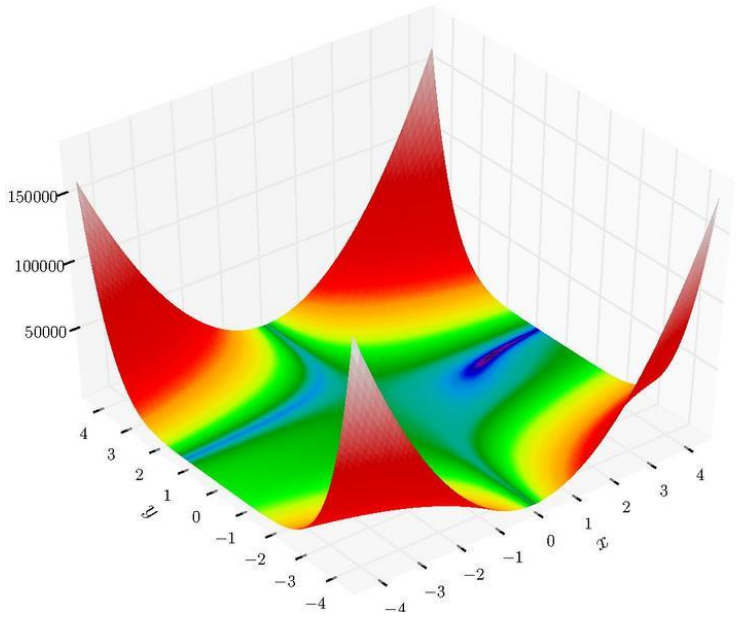

Рис.3.3. Функция Beale

Для функции McCormick (рис.3.4):

$$f(x,y) = \sin(x+y) + (x-y)^2 - 1.5x + 2.5y + 1,$$

где $-10 \leq x,y \leq 10$, минимум был локализован в $f(-0.547,-1.548) = -1.913$ за 6.7 сек.

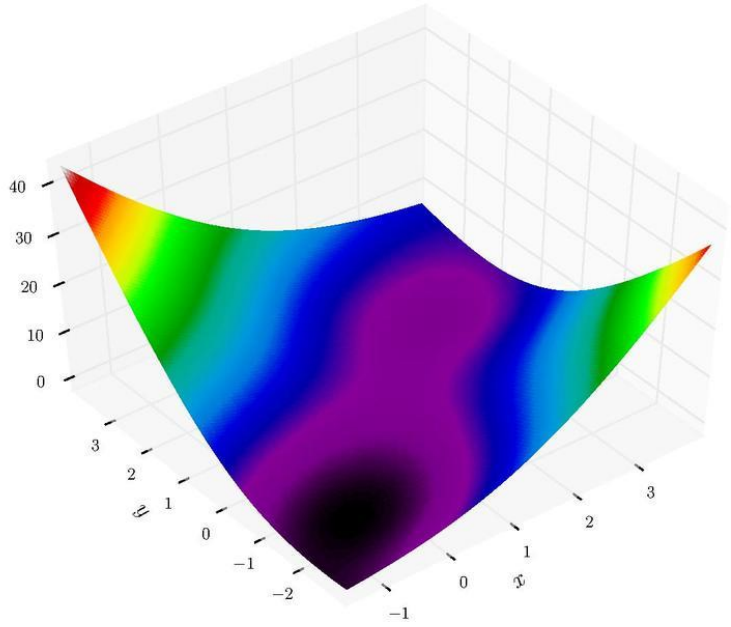

Рис.3.4. Функция McCromick

Для функции Shubert (рис.3.5)

$$f(x) = \left(\sum_{i=1}^{5} i \cdot \cos\big((i+1)x_1 + i\big)\right)\left(\sum_{i=1}^{5} i \cdot \cos\big((i+1)x_2 + i\big)\right),$$

где $-10 \le x_i \le 10, i = 1, 2$, минимум равный -186.7309 был локализован за 6.1 сек на 18 брусах:

$[-7.0835,\ 4.8580]$, $[-7.0835, -7.7083]$,

$[-7.7083, -7.0835]$, $[-7.7083,\ \ 5.4828]$,

$[-1.4251, -7.0835]$, $[-1.4251, -0.8003]$,

$[-1.4251, 5.4828]$, $[-7.7083, -0.8003]$,

$[4.8580\ 5.4828]$, $[-7.0835,\ \ 1.4251]$,

$[-5.4828, 4.8580]$, $[-0.8003, -1.4251]$,

$[-0.8003, -7.7083]$, $[-0.8003,\ \ 4.8580]$,

$[5.4828, -1.4251]$, $[5.4828, -7.7083]$,

$[4.8580, -7.0835]$, $[4.8580, 0.8003]$,

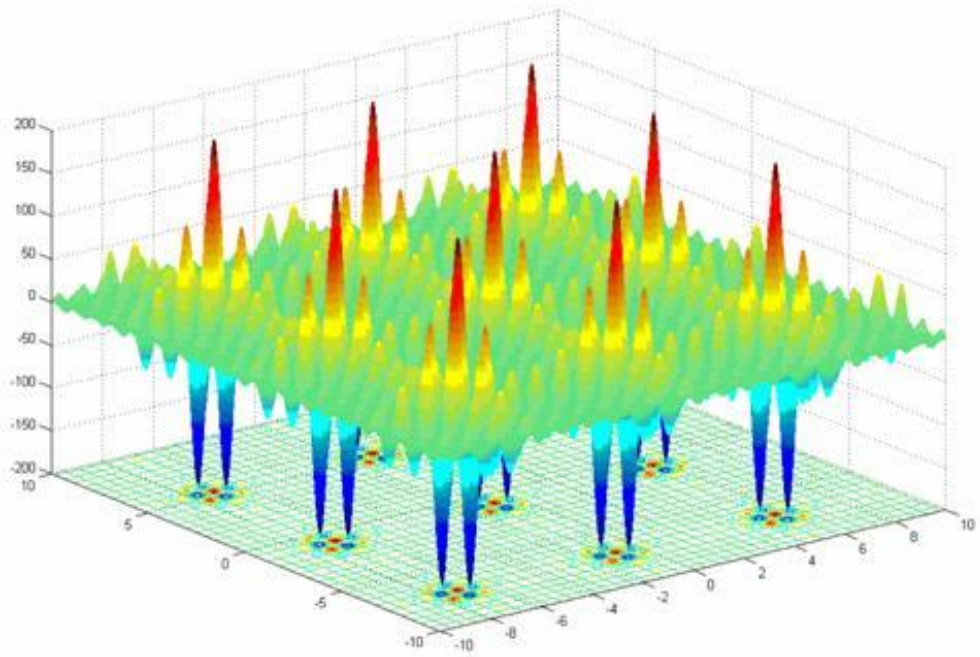

Рис 3.5. Функция Shubert

Все проведенные тесты на корректность вычисления глобального оптимума заданных функций были успешно пройдены.

3.2 Экспериментальное оценивание эффективности распараллеливания

Для оценивания эффективности распараллеливания использовался тот же кластер, что и при тестировании корректности вычислений. Проводилось сравнение времени работы последовательного алгоритма GOptimum и времени работы распределенной решателя задач, путем вычисления коэффициента ускорения:

$$S(k) = T(1)/T(k), \tag{3.1}$$

где $k -$ количество узлов распределенной системы.

В качестве первой тестовой функции была выбрана функция Розенброка:

$$f(x) = \sum_{i=1}^{n-1} \left(100 \cdot \left(x_{i+1} - x_i^2\right)^2 + (x_i - 1)^2\right),$$

где $1000 \leq x_i \leq 1000$. Эксперимент проводился для семейства функций с разными $n = (10,\ 20,\ 30,\ 40,\ 50,\ 60,\ 70,\ 80,\ 90,\ 100)$, что позволяло масштабировать объем вычислений и смотреть каким образом реагируют на изменение размерности последовательный и распределенный алгоритм. В том числе количество вычислительной работы было обусловлено необходимой точностью вычислений, которая с уменьшением порождает необходимость исследовать большее количество брусов. Для всех n найденный глобальный минимум равен $= 0$. В таблице 3.1 приведены результаты экспериментов для всех n и точностью поиска $1e - 10$. На рисунке 3.6 изображен график коэффициента ускорения поиска глобального оптимума с точностью $1e - 10$.

Таблица 3.1 Эксперименты поиска глобального оптимума для функции Розенброка с точностью 1e-10.

n	Время последовательного поиска, сек	Время поиска на 2-х узлах, сек	Время поиска на 3-х узлах, сек	Время поиска на 4-х узлах, сек	Коэффициент ускорения для 2-х узлов	Коэффициент ускорения для 3-х узлов	Коэффициент ускорения для 4-х узлов
10	1.34	2.47	3.61	4.52	0.54251	0.371191	0.29646
20	2.13	2.89	4.36	5.17	0.737024	0.488532	0.411992
30	3.15	3.34	5.63	5.43	0.943114	0.559503	0.58011
40	4.45	4.15	6.3	6.12	1.072289	0.706349	0.727124
50	6.23	5.43	7.82	6.93	1.14733	0.796675	0.89899
60	8.43	7.19	8.36	7.54	1.172462	1.008373	1.118037
70	10.61	9.37	9.29	8.74	1.132337	1.142088	1.213959
80	12.89	10.89	10.66	10.27	1.183655	1.209193	1255112
90	14.15	12.1	12.31	11.76	1.169421	1.149472	1247795
100	15.65	13.96	15.02	12.05	1.12106	1.116262	1298755

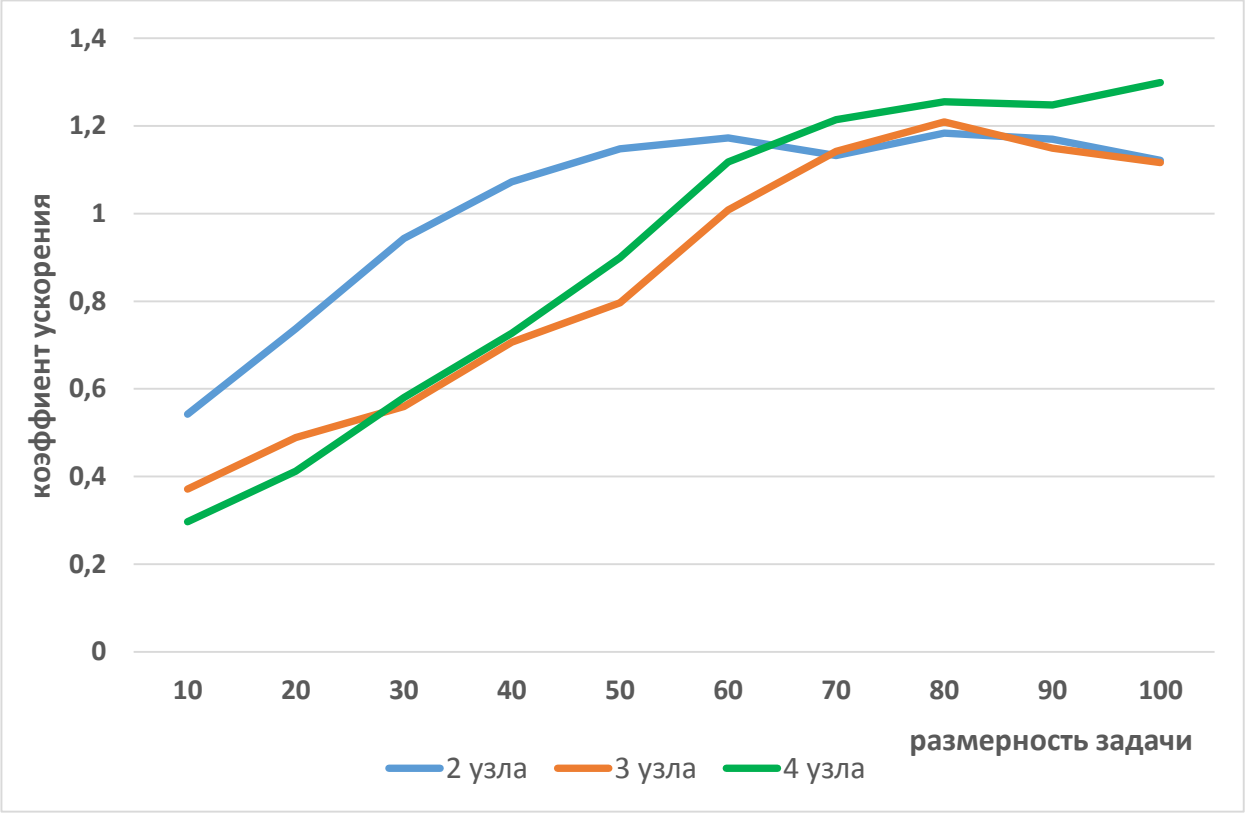

Рис.3.6. График коэффициентов ускорения поиска глобального оптимума на 2-х, 3-х, 4-х узлах с точностью 1e-10.

В таблице 3.2 приведены результаты экспериментов для всех n и точностью поиска $1e-20$. На рисунке 3.7 изображен график коэффициента ускорения поиска глобального оптимума с точностью 1e-20.

Таблица 3.2 Эксперименты поиска глобального оптимума для функции Розенброка с точностью 1e-20.

n	Время последовательного поиска, сек	Время поиска на 2-х узлах, сек	Время поиска на 3-х узлах, сек	Время поиска на 4-х узлах, сек	Коэффициент ускорения для 2-х узлов	Коэффициент ускорения для 3-х узлов	Коэффициент ускорения для 4-х узлов
10	1.76	3.12	4.2	5.01	0.564103	0.419048	0.351297
20	2.86	3.97	4.96	5.25	0.720403	0.576613	0.544762
30	4.34	4.22	5.7	5.86	1.028436	0.761404	0.740614
40	5.36	5.1	6.83	6.64	1.05098	0.784773	0.807229
50	7.52	6.48	7.25	7.13	1.160494	1.037241	1.054698
60	8.97	7,64	8.66	7.78	1.174084	1.035797	1.152956
70	11.37	9.69	9.41	8.74	1.173375	1.208289	1.300915
80	13.5	11.48	10.73	9.13	1.175958	1.258155	1.478642
90	16.85	12.6	13.35	11.68	1.337302	1.262172	1.442637
100	18.39	14.27	15.4	12.45	1.288718	1.194156	1.477108

Из графиков видно, что два узла эффективно использовать при $30 \leq n \leq 60$, после этого наступает насыщение работы алгоритма. Переломный момент для решения на 3-х и 4-х узлах наступает после $n = 60$, когда коэффициент ускорения становится больше 1. Наиболее эффективным показало себя решение задачи на 4 узлах при размерности больше 60. Время решения задачи при $n = 80$ и точностью $1e-20$ у распределенного алгоритма на 4-х узлах в 1.5 раза меньше чем у последовательного.

Рис.3.7. График коэффициентов ускорения поиска глобального оптимума на 2-х, 3-х, 4-х узлах с точностью 1e-20.

Следующая тестовая функция - Pathological Function:

$$f(x) = \sum_{i=1}^{m-1} 0.5 + \left(\frac{sin^2\left(\sqrt{100x_{i+1}^2 + x_i^2}\right) - 0.5}{1 + 0.001(x_i^2 - 2x_i x_{i+1} + x_{i+1}^2)^2} \right),$$

где $1000 \leq x_i \leq 1000$. Эксперимент проводился для семейства функций с разными $m = (10,\ 20,\ 30,\ 40,\ 50,\ 60,\ 70)$. Для всех m найденный глобальный минимум равен 0.

В таблице 3.3 приведены результаты экспериментов для всех m и точностью поиска $1e - 10$. На рисунке 3.8 изображен график ускорения поиска глобального оптимума на 2-х, 3-х, 4-х узлах с точностью $1e - 10$.

Таблица 3.3 Эксперименты поиска глобального оптимума для функции Pathological с точностью 1e-10.

m	Время последовательного поиска, сек	Время поиска на 2-х узлах, сек	Время поиска на 3-х узлах, сек	Время поиска на 4-х узлах, сек	Коэффициент ускорения для 2-х узлов	Коэффициент ускорения для 3-х узлов	Коэффициент ускорения для 4-х узлов
10	133	19.6	16.4	14.88	0.678571	0.810976	0.893817
20	45.85	40.16	37.82	34.19	1.141683	1.212322	1.341035
30	179.44	143.57	134.55	119.4	1.249843	1.333631	1.502848
40	286.41	221.5	208.13	177.38	1.293047	1.376111	1.614669
50	412.67	315.81	296.09	243.67	1.306703	1.393732	1.693561
60	678.12	479.22	422.73	338.22	1.415049	1.604144	2.004967
70	911.45	591.41	510.53	447.76	1.541147	1.785302	2.035577

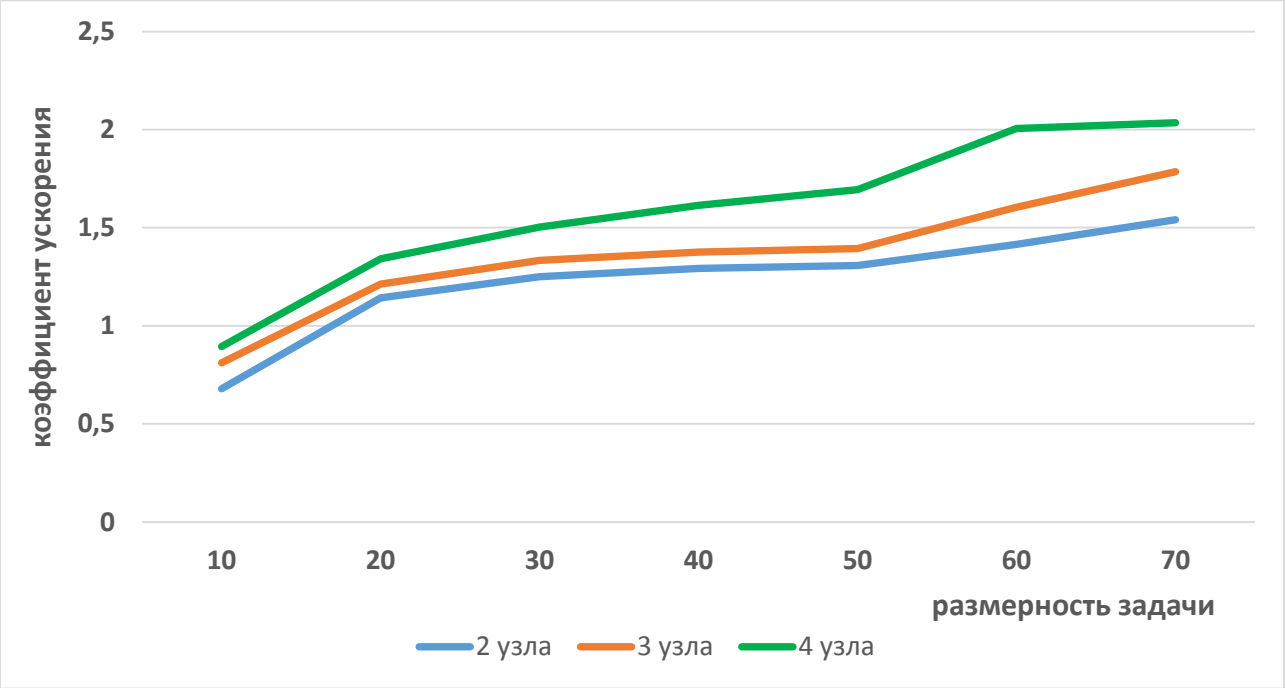

Рис.3.8. График коэффициентов ускорения поиска глобального оптимума на 2-х, 3-х, 4-х узлах с точностью 1e-10.

Из графика видно, что при $m \geq 20$ эффективнее использовать распределенный алгоритм, причем с увеличением m эффективность увеличивается. На 4-х узлах при размерности ≥ 60 время описка оптимума уменьшается в 2 раза по сравнению с последовательным алгоритмом. Это доказывает, что использование распределенного решателя задач наиболее перспективно, когда сложность задачи высока.

Выводы к третьей главе

1. Проведенные тестовые запуски распределённого решателя задач глобальной оптимизации на эталонных целевых функция показали корректность поиска глобального оптимума.
2. Проведенные эксперименты на нескольких целевых функциях показали эффективность использования распределённого решателя задач.

Заключение

Таким образом, цель работы, состоящая в создании распределенного решателя задач глобальной оптимизации функции многих переменных достигнута, а именно получены следующие результаты.

1. Проведен обзор программных инструментов для решения задач глобальной оптимизации и систем построения распределенных вычислений. В качестве основы для реализации распределенного решателя задач глобальной оптимизации были выбраны приложение GOptimum и фреймворк Apache Ignite, как наиболее подходящие для достижения поставленной цели.
2. После исследования механизмов и инструментов, предоставленных приложением GOptimum и фреймворка Apache Ignite, на их основе было спроектировано и разработано приложение, распределённый решатель задач глобальной оптимизации.
3. Проведены вычислительные эксперименты для тестирования вычислительной корректности. Все тесты были успешно пройдены. Также проводились эксперименты для выяснения условий эффективного использования решателя задач глобальной оптимизации. Было установлено, что для эффективного использования задача глобальной оптимизации должна быть достаточно сложной. Увеличение количества вычислительных узлов, повышает эффективность поиска глобального оптимума на таких задачах.

Разработанное приложение является готовым программным продуктом, может использоваться для решения задач глобальной оптимизации на многомерных, многоэкстремальных функциях.

Результаты работы докладывались на Третьей региональной конференции «Мой выбор наука!» (г. Барнаул, апрель 2016), в секции «Интеллектуальный анализ данных. Информационные системы и технологии» были удостоены диплома первой степени.

Библиографический список

1. Гаганов, А.А. О сложности вычисления интервала значений полинома от многих переменных. Кибернетика. – 1985.
2. Kreinovich, V., Kearfott R.B. Beyond convex. Global optimization is feasible onlyfor convex objective functions: a theorem // Journal of Global Optimization. – 2005. – Vol. 33, No. 4. – P. 617-624.
3. Neumaier, A. Global Optimization Software – [Электронный ресурс]. – URL: http://www.mat.univie.ac.at/~neum/glopt/software_g.html.
4. Interval and Related Software – [Электронный ресурс]. – URL: http://www.cs.upc.edu/~robert/mirror/interval-comp/intsoft.html.
5. Панов, Н.В. Разработка рандомизированных алгоритмов в интервальной глобальной оптимизации. Диссертация кандидата физ.-мат. наук Новосибирск. – 2012.
6. Шарый, С.П. Конечномерный интервальный анализ – [Электронный ресурс]. – URL: http://www.nsc.ru/interval/Library/InteBooks/SharyBook.pdf
7. Шарый, С.П. Рандомизированные алгоритмы в интервальной глобальной оптимизации // Сиб. Журнал Вычисл. Матем. – 2008. –Т. 11, №4. – С. 457–474.
8. Звонков, В.Б., Попов, А.М., Сравнительное исследование классических методов оптимизации и генетических алгоритмов // Вестник СибГАУ. – 2013. – №4 (50). – С. 23-27.
9. Панов, Н.В. Комбинирование точечных и интервальных методов поиска глобального оптимума // XII Всероссийская конференция молодых ученых по математическому моделированию и информационным технологиям. Россия, г. Новосибирск, 3-6 октября 2011г. Дополненные тезисы докладов, – С. 55-56.
10. Коротченко А.Г. Об одном алгоритме поиска наибольшего значения одномерных функций. // ЖВМ и МФ – 1978. –Т. 18, №3. – С. 563-573.

11. Kushner H. A. New Method of Locating the Maximum Point of an Arbitrary Multipeak Curve in the Presence of Noise. // Transactions ASME, Ser. D.J. Basic Eng., – 1964. – Vol. 86, No. 1. – P. 97–106.
12. Ершов А.Р., Хасимов О.В. Автоматическая глобальная оптимизация // Труды XIV Байкальской международной школы-семинара «Методы оптимизации и их приложения», Иркутск, Байкал, 2-8 июля 2008 года. Т.1 «Математическое программирование». – Иркутск, ИСЭМ СО РАН, – 2008. – С. 235–244.
13. Шарый С.П. Стохастические подходы в интервальной глобальной оптимизации // Труды XIII Байкальской международной школы-семинара «Методы оптимизации и их приложения», Иркутск-Северобайкальск, 2-8 июля 2005 года. Т. 4 «Интервальный анализ». – Иркутск, ИСЭМ СО РАН, – 2005. С. 85–105.
14. Панов Н.В., Шарый С.П., Развитие стохастических подходов в интервальной глобальной оптимизации // VIII Всероссийская конференция молодых ученых по математическому моделированию и информационным технологиям, Новосибирск, 27-29 ноября 2007г., Тезисы докладов – Новосибирск, – 2007. – С. 22.
15. Панов Н.В. Развитие стохастических подходов в интервальной глобальной оптимизации. Интервальный генетический алгоритм // Международная конференция «Современные проблемы математического моделирования и вычислительных технологий - 2008», Красноярск, 18-24 августа 2008 г. Тезисы докладов. – Красноярск: СФУ. – С. 31-32.
16. Азенкотт Р. Процедура «отпуска» // Труды семинара Н. Бурбаки за 1988 г. – М.: Мир, 1990. – С. 235-251.
17. Жиглявский А.А., Жилинскас А.Г., Методы поиска глобального экстремума. – М.: Наука, 1991.
18. Metropolis N., Rosenbluth M., Equation of State Calculations by Fast Computing Machines // J. Chem. Phys. – 1953. – Vol.21. – P. 1087.

19. Sahinidis, N.V., BARON 14.4.0: Global Optimizations of Mixed-Integer Nonlinear Programs – [Электронный ресурс].
– URL: http://archimedes.cheme.cmu.edu/?q=baron
20. Kearfott R.B., Download GlobSol – [Электронный ресурс].
– URL: http://interval.louisiana.edu/GlobSol/download_GlobSol.html
21. Hu C., Kearfott B., A Parallel Software Package for Nonlinear Global Optimization., – [Электронный ресурс].
– URL: http://www.mat.univie.ac.at/~neum/glopt/mss/HuKX00.pdf
22. Panov N.V., Global Optima Solver. – [Электронный ресурс].
– URL: https://github.com/nvpanov/goptimum
23. Hickey T., Qiu Z., Interval Constraint Plotting for Interactive Visual Exploration of Implicitly Defined Relations. – [Электронный ресурс].
– URL: http://interval.sourceforge.net/interval/java/ia_math/README.html
24. Радченко Г.И., Распределенные вычислительные системы. – Челябинск. – Фотохудожник, 2012. – 184 с.
25. HazelCast – [Электронный ресурс]. – URL: https://hazelcast.com/
26. Welcome to Apache Hadoop! – [Электронный ресурс].
– URL: http://hadoop.apache.org/
27. Apache Ignite In-Memory Data Fabric – [Электронный ресурс].
– URL: https://ignite.apache.org/
28. Apache Maven Project – [Электронный ресурс].
– URL: https://maven.apache.org/
29. What is Ignite – [Электронный ресурс].
– URL: http://apacheignite.readme.io/docs
30. Momin J., Xin-She Y., A literature Survey of Benchmark Functions For Global Optimizations Problems // Int. Journal of Mathematical Modelling and Numerical Optimization, Vol. 4. – 2013. – P. 150–194.
31. Test functions for optimizations – [Электронный ресурс].

– URL: https://en.wikipedia.org/wiki/Test_functions_for_optimization

32. Test functions for Unconstrained Global Optimizations – [Электронный ресурс].–URL:http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO_files/Page364.htm
33. Handbook of Test Problems in Local and Global Optimization – [Электронный ресурс]. –URL: http://titan.princeton.edu/TestProblems/
34. Пратт, Т., Зелковиц, М., Языки программирования: разработка и реализация. – 4-е издание. – Питер, 2002. – 688 с.

Приложение

Исходные коды приложения
«Распределенный решатель задач глобальной оптимизации»