Московский государственный университет имени М.В. Ломоносова
Факультет Вычислительной математики и кибернетики
Кафедра Суперкомпьютеров и квантовой информатики

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Разработка и исследование параллельного алгоритма дифференциальной эволюции для различных классов задач

Выполнил:
студент 423 группы
Полеводов Андрей Сергеевич

Научный руководитель:
к.ф-м.н., доцент
Попова Нина Николаевна

Москва, 2016

Оглавление

Аннотация

В данной работе рассматривается алгоритм дифференциальной эволюции, его параллельная реализация и исследование. Для тестирования определяется набор тестовых функций, определяются их параметры. В ходе работы реализуется классическая островная модель алгоритма. Проводится исследование зависимости времени работы алгоритма от различных параметров алгоритма. Проводится исследование поведения алгоритма на задачах с большой размерностью, в результате которого разрабатывается специальная параллельная модель для задач с большой размерностью - модель декомпозиции по группам генов. Проводится сравнительный анализ островной модели и модели декомпозиции по группам генов, из которого делаются выводы о точности и времени работы алгоритма.

1. Введение

Во многих прикладных задачах возникает подзадача поиска некоторой оптимальной конфигурации системы. Она может быть описана по-разному, но математически ее можно описать следующим образом[1]:

$$arg\,min_{x \in X} f(x) \quad \text{или} \quad arg\,max_{x \in X} f(x)$$

То есть нужно найти ту конфигурацию (набор параметров), что отражается в аргументе функции f, при которой некоторая система, которую отражает функция, будет оптимальна: функция f будет либо максимизирована, либо минимизирована, в зависимости от задачи.

Некоторые функции обладают известным экстремальным значением, для некоторых можно найти экстремум аналитически, используя аппарат математического анализа: например, функция $f(x) = x^2$ имеет экстремум в точке $x = 0$ и $f(0) = 0$, который можно найти продифференцировав данную функцию, чтобы найти точку перегиба. Для некоторых функций экстремум можно найти с помощью методов оптимизации: например, метод динамического программирования для задач дискретной оптимизации, метод градиентного спуска для задач непрерывной оптимизации. Для некоторых функций экстремум можно найти с помощью численных методов, например, аппроксимируя функцию более простой и находя экстремум у аппроксимированной функции.

Все вышеуказанные методы накладывают те или иные требования на функцию, такие как унимодальность (наличие одного экстремума), дифференцируемость функции, непрерывность функции. Для произвольной функции данные требования зачастую не выполняются. Любую задачу оптимизации можно решить перебором, но пространство аргументов оптимизируемой функции почти всегда очень велико, поэтому перебор аргументов - достаточно трудоемкий процесс, поэтому необходима оптимизация перебора. Поэтому были разработаны эволюционные алгоритмы, которые на базе представлений о естественном отборе реализуют эволюцию множества аргументов функции, оптимум которой необходимо найти.

[1] Это не является формальной постановкой, формально задача оптимизации описана в главе "Задача оптимизации"

1.1 Задача оптимизации

Задача оптимизации состоит в определении наилучших, в некотором смысле, структуры или значений параметров объектов. Такая задача называется оптимизационной. Если оптимизация связана с расчётом оптимальных значений параметров при заданной структуре объекта, то она называется *параметрической оптимизацией*. Задача выбора оптимальной структуры является *структурной оптимизацией*.

Стандартная математическая задача оптимизации формулируется таким образом. Среди элементов χ, образующих множества $\mathbb{X}$, найти такой элемент χ^*, который доставляет оптимальное значение $f(\chi^*)$ заданной функции $f(\chi)$. Для того, чтобы корректно поставить задачу оптимизации, необходимо задать:

1. *Допустимое множество* — множество $\mathbb{X} = \{x | g_i(x) \leq 0, i = 1, \dots, m\} \subset \mathbb{M}$ ($\mathbb{M}$ - пространство всех возможных аргументов, $g_i(x)$ - ограничения)
2. *Целевую функцию* — отображение $f: \mathbb{X} \to \mathbb{R}$
3. *Критерий поиска* (max или min).

Тогда решить задачу $f(x) \to \text{opt}_{x \in \mathbb{X}} f(x)$ означает одно из:

1. Показать, что $\mathbb{X} = \emptyset$.
2. Показать, что целевая функция $f(x)$ не ограничена.
3. Найти $x^* \in \mathbb{X}$: $f(x^*) = \text{opt}_{x \in \mathbb{X}} f(x)$.

Если оптимизируемая функция не является выпуклой, то часто ограничиваются поиском локальных минимумов и максимумов: точек x_0 таких, что всюду в некоторой их окрестности $f(x) \geq f(x_0)$ для минимума и $f(x) \leq f(x_0)$ для максимума.

Если допустимое множество $\mathbb{X} = \mathbb{M}$, то такая задача называется *задачей безусловной оптимизации*, в противном случае — *задачей условной оптимизации*.

В качестве тестовых задач будут рассматриваться задачи безусловной оптимизации в пространстве векторов над вещественным полем ($\mathbb{R}^n$).

1.2 Эволюционные алгоритмы

Сначала нужно обозначить основные определения эволюционных алгоритмов и их структурные единицы, так как алгоритм дифференциальной эволюции является эволюционным алгоритмом. Эволюционные алгоритмы - направление в искусственном интеллекте, которое использует и моделирует

процессы естественного отбора. Структурными единицами алгоритма являются:

- *Популяция* (*поколение*) - набор хромосом: $\{C_1, \dots, C_n\}$.
- *Хромосома* (*особь/вектор*) - объект, являющийся аргументом фитнесс-функции: $C = \arg(f)$. Чаще всего хромосома представляется в виде вектора - набора генов $(x_1, \dots, x_d)$. Например, хромосомой может быть бинарный вектор для задачи о рюкзаке или вектор вещественных чисел для функции от вещественных аргументов.
- *Ген* - элемент хромосомы, компонент вектора оптимизируемой функции x_i.
- *Фитнесс-функция* (*целевая функция/приспособленность*) - функция, для которой необходимо по условию задачи найти оптимум: $f: C \to F$ $(F = \mathbb{N}, \mathbb{Q}, \mathbb{R} \dots)$.

По аналогии с соответствующими биологическими понятиями, есть еще множество базовых элементов алгоритма, но я не буду сильно вдаваться в терминологию. В таких алгоритмах сначала инициализируется первая популяция и далее строятся следующие популяции с помощью следующих операторов над хромосомами из популяции:

- *Оператор мутации* - стохастический оператор, переводящий одну или несколько хромосом в измененные случайным образом одну или несколько хромосом:

  $$C^n \xrightarrow{Mutation} C^k$$

  Например, бинарный вектор (101) может перейти в вектор (111), если оператор состоит в случайном замещении одной координаты на другую.
- *Оператор кроссовера* (кроссинговера/скрещивания) - оператор, переводящий набор хромосом в другой набор хромосом по специальному правилу, возможно с некоторым стохастическим воздействием:

  $$C^n \xrightarrow{Crossover} C^k$$

  Причем хромосомы в левой части называются родителями (предками), а в правой - потомками. Чаще всего, по аналогии с биологическим скрещиванием $n = 2$, а $k = 1$ или 2, что означает наличие у двух родителей одного или двух потомков. Например, бинарные векторы (101) и (011) могут быть переведены в вектор (110), в данном случае оператор кроссовера - бинарная операция XOR.

- *Оператор селекции* - оператор, производящий отбор хромосом в следующую популяцию либо для последующего применения другого оператора.

Несмотря на схожесть определений операторов кроссовера и мутации, данные операторы служат для различных целей: оператор мутации необходим для "генетического разнообразия" популяции, чтобы предотвратить "сваливание" алгоритма в локальный оптимум, а оператор кроссовера необходим для сходимости алгоритма: чтобы на новой итерации получались хромосомы, фитнесс-функция которых ближе к оптимуму, чем фитнесс-функция предков.

Отбор хромосом в некотором наборе определяется с помощью оператора селекции. Обычно применяется селекция по значению фитнесс-функции: например, если нам нужно найти минимум функции $x_1 + x_2$ то хромосома (0,0) будет "лучше" хромосомы (1,1).

Алгоритм останавливается в зависимости от одного или нескольких условий останова:

- Нахождение глобального, либо субоптимального решения.
- Исчерпание числа поколений, отпущенных на эволюцию.
- Исчерпание времени, отпущенного на эволюцию.
- Неизменность оптимума в течение нескольких поколений.

Отмечу, что первое условие останова подходит лишь для тестовых задач, для которых заранее известно оптимальное либо субоптимальное решение, остальные условия останова подходят для произвольной задачи. После завершения работы алгоритма, хромосома с наиболее оптимальным значением фитнесс-функции выбирается в качестве ответа.

1.3 Оператор кроссовера

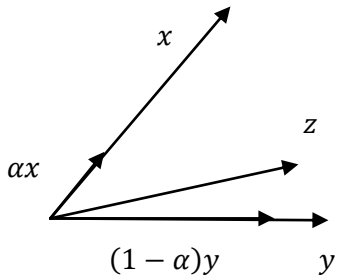
Рассмотрим поподробнее операторы, о которых сказано выше, а именно, рассмотрим виды оператора кроссовера:

- *Одноточечный кроссинговер.* Он моделируется следующим образом: пусть имеются две родительские хромосомы $(x_1, \dots, x_n)$ и $(y_1, \dots, y_n)$. Случайным образом определяется точка внутри хромосомы (точка разрыва), в которой обе хромосомы делятся на две части и обмениваются ими. Такой тип кроссинговера называется одноточечным, так как при нем родительские

хромосомы разделяются только в одной случайной точке. Пример кроссовера показан на схеме ниже:

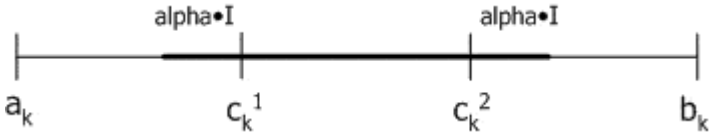
Родители		Потомки
$(x_1, \dots, x_k, x_{k+1}, \dots, x_n)$	$\mapsto$	$(x_1, \dots, x_k, y_{k+1}, \dots, y_n)$
$(y_1, \dots, y_k, y_{k+1}, \dots, y_n)$		$(y_1, \dots, y_k, x_{k+1}, \dots, x_n)$

- *Многоточечный кроссинговер.* Случайно выбирается m точек разрыва и с каждой последовательно проводится одноточечный кроссинговер.
- *Однородный кроссинговер.* Для каждого i из интервала [1,n] (n - количество генов) происходит обмен геном между предками с некоторой вероятностью, который является параметром оператора. На выходе либо выбирается 1 потомок из измененных предков, либо выбираются оба измененных предка.
- *Арифметический кроссинговер.* Данный кроссинговер характерен для непрерывных задач и строится следующим образом: $z = \alpha x + (1 - \alpha)y$, где x, y - родители, а z - потомок. α - параметр кроссинговера, может быть многомерным, каждая компонента задана из интервала [0,1]. На схеме ниже показан одномерный случай:



- *Геометрический кроссинговер.* Также придуман для задач непрерывной оптимизации. Он аналогичен арифметическому кроссоверу, и строится следующим образом: $z = x^{\alpha} + y^{(1-\alpha)}$, где x, y - родители, а z - потомок. α - параметр кроссовера, может быть многомерным, каждая компонента задана из интервала [0,1].
- *Плоский кроссинговер.* Подходит исключительно для задач непрерывной оптимизации и является обобщением однородного кроссинговера: для каждого k из интервала [1,n] (n - количество генов) случайным образом выбирается число из интервала $[c_k^1, c_k^2]$, где c_k^1 и c_k^2 - гены предков.
- *BLX-alpha кроссинговер.* Является обобщением плоского кроссинговера, и отличается тем, что возможен выбор гена из некоторой

внешней окрестности интервала $[c_k^1, c_k^2]$, что зависит от параметра α. Пусть $I = |c_k^1 - c_k^2|$ и пусть $c_k^1 < c_k^2$, тогда результат выбирается случайно из интервала $[c_k^1 - \alpha I, c_k^2 + \alpha I]$ как показано на схеме:



- *DE кроссинговер.* Кроссинговер, придуманный Шторном и Прайсом и использованный в алгоритме дифференциальной эволюции, является частным случаем однородного кроссинговера. Сначала случайным образом генерируются числа j и l из интервалов [0,n-1] и [1,n] соответственно. Далее результат высчитывается следующим образом:

$$z_k = \begin{cases} c_k^1, & for\ k = |j|_n, |j+1|_n, \dots, |j+l|_n \\ c_k^2, & for\ all\ other\ k\ in\ [0,n] \end{cases}, \text{где } ||_n$$

$-$ операция взятия числа по модулю n

То есть происходит циклическая замена l генов начиная с позиции j.

Конечно, здесь представлены не все операторы кроссинговера: можно как-либо специфицировать эти операторы, можно их комбинировать, можно придумать свой оператор, но их суть одно и та же: обеспечить сходимость алгоритма. Существует также множество общих операторов мутации для эволюционных алгоритмов, но в алгоритме дифференциальной эволюции оператор мутации имеет специальный вид и о нем будет сказано ниже.

1.4 Алгоритм дифференциальной эволюции

Алгоритм дифференциальной эволюции (Differential Evolution, DE) - стохастический итерационный алгоритм многомерной математической оптимизации, являющийся одним из алгоритмов эволюционных вычислений и использующий идеи генетических алгоритмов. Был разработан и опубликован в 1995 году.

Алгоритм относится к классу прямых методов оптимизации, то есть он требует только возможность вычислить значение целевой функции. Предназначен для нахождения глобального оптимума функций от многих переменных, которые могут быть недифференцируемы, мультимодальны (иметь несколько локальных экстремумов). Хорошо подходит для непрерывной оптимизации.

Алгоритм отличается от генетических алгоритмов тем, что мутация происходит с использованием хромосом текущий популяции, в то время как в генетических алгоритмах мутацию производят сгенерированные

случайным образом числа. То есть, в генетических алгоритмах мутация "порождается извне", а в дифференциальной эволюции мутация порождается элементами текущей популяции, что наделяет образ оператора мутации свойством локальности (образ отстоит от прообраза не более, чем на некоторое число), что в некоторых случаях может быть полезно в решении.

В классическом алгоритме (предложенном Шторном и Прайсом) итерация происходит следующим образом:

1. Выбирается очередная хромосома x из предыдущей популяции, назовем ее перебираемой хромосомой или перебираемым вектором.
2. Для нее случайным образом выбираются 3 особи: v_0, v_1, v_2 и строится пробный вектор v по следующему правилу:
$$v = v_0 + F * (v_1 - v_2)$$
где F - сила мутации - положительная константа, поэтому для особей должны быть определены операции сложения, вычитания и умножения на константу. Пример построения пробного вектора показан на рисунке ниже:

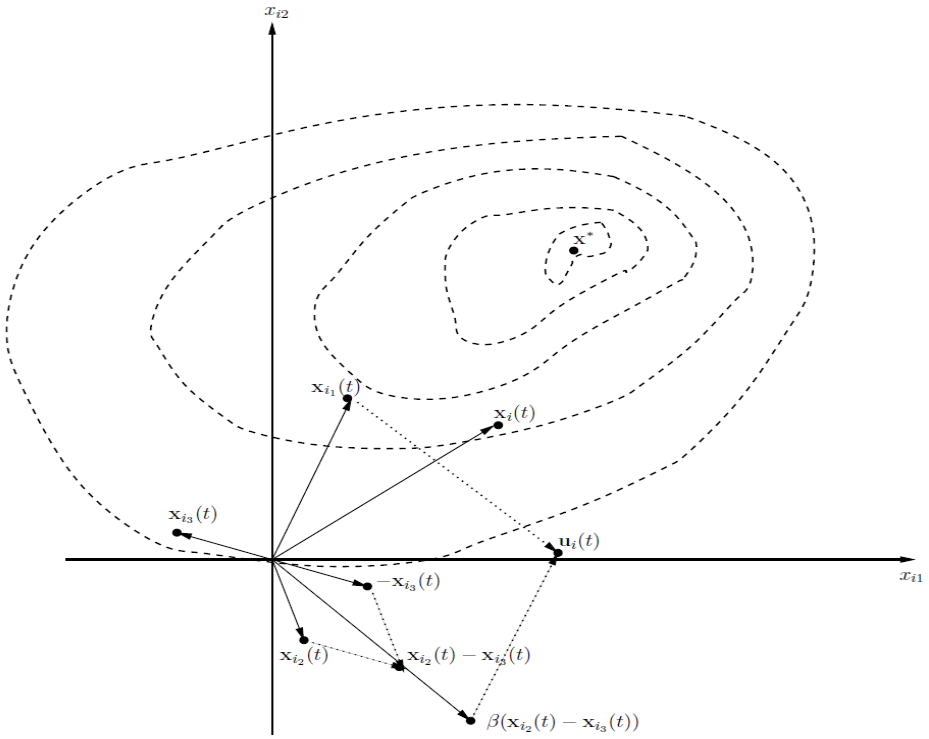


Рисунок 1. Построение пробного вектора (на рисунке - $\boldsymbol{u_i(t)}$)

3. Над особями v и x выполняется оператор кроссовера, в результате которого получается особь y.

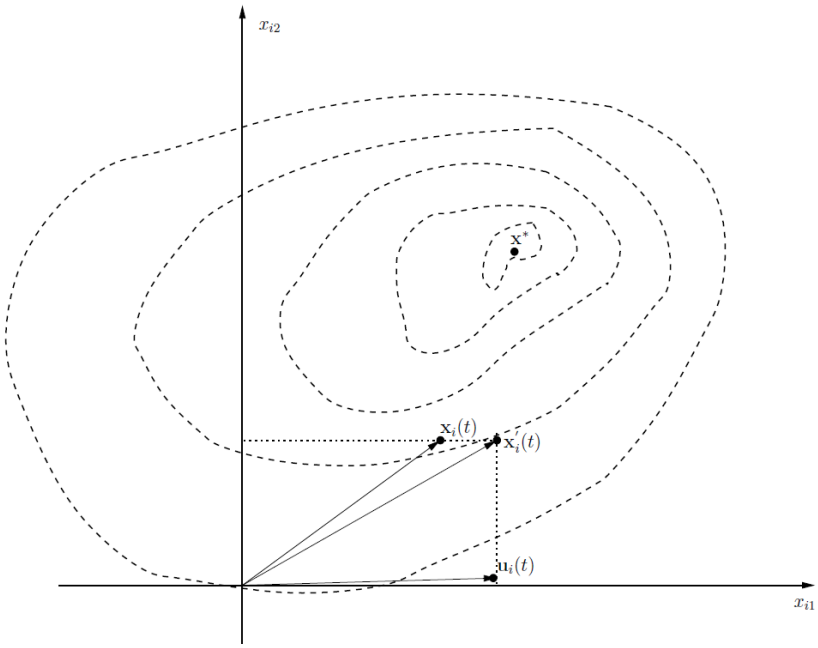


Рисунок 2. Однородный кроссовер для пробного вектора и текущей особи (на рисунке - $\boldsymbol{u_i(t)}$ и $\boldsymbol{x_i(t)}$)

4. С помощью оператора селекции из особей x и y выбирается наиболее подходящая z, и она становится частью новой популяции.
5. Алгоритм останавливается при выполнении одного или нескольких условий останова:
   - Нахождение глобального, либо субоптимального решения.
   - Исчерпание числа поколений, отпущенных на эволюцию.
   - Исчерпание времени, отпущенного на эволюцию.
   - Неизменность оптимума в течение нескольких поколений.

Формально итерацию классической версии алгоритма можно описать следующим образом:

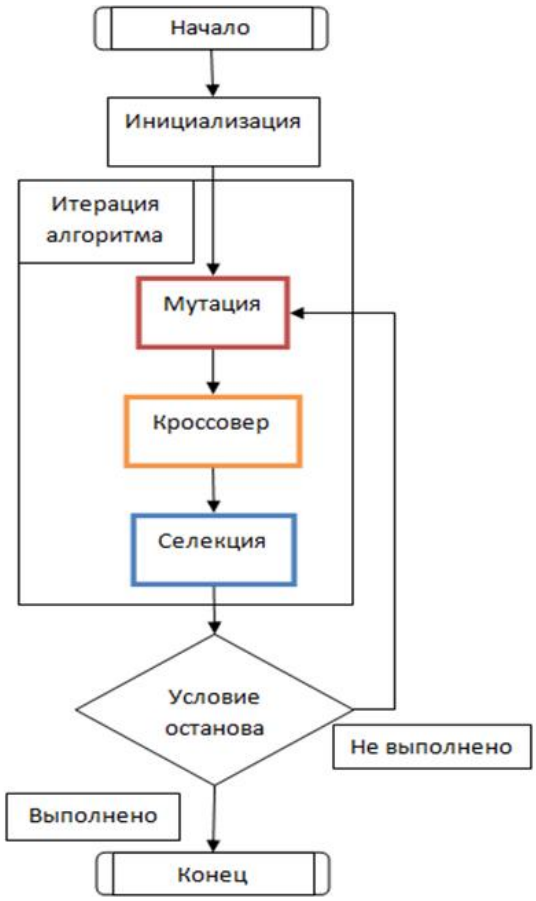
Пусть D - размерность задачи, NP - размер популяции, F - вещественная положительная константа, x_i^g - i-й элемент поколения g, $v[j]$ - j-й ген хромосомы v, r_1, r_2, r_3 - 3 случайных целых числа, выбранных из интервала [0,NP-1], s, l - случайные числа, выбранные из интервалов [0,D-1] и [1,D] соответственно, тогда итерация алгоритма (построения поколения $g+1$ из поколения g) происходит следующим образом:

- Для каждого $x_i^g, i \in [0, NP-1]$
- Проводится оператор мутации, строится пробный вектор:
$$v = x_{r_1}^g + F * (x_{r_2}^g - x_{r_3}^g)$$
- Проводится оператор кроссовера:

$$y[k] = \begin{cases} v[k], & for\ k = |s|_D, |s+1|_D, \dots, |s+l|_D \\ x_i^g[k], & for\ all\ other\ k\ in\ [0, D] \end{cases}$$, где $||_D$ – операция взятия числа по модулю D

- Если $f(y) < f(x_i^g)$ (если задача состоит в поиске минимума, иначе - $f(y) > f(x_i^g)$), то $x_i^{g+1} = y$, иначе $x_i^{g+1} = x_i^g$

На рисунке ниже показаны общая блок-схема последовательного алгоритма и псевдокод оригинального алгоритма дифференциальной эволюции. На блок-схеме цветами показаны операторы алгоритма.




```
1  P(i) - популяция iй итерации (поколение номер i)
2  NP - число хромосом в популяции (NP = |P(i)|)
3  N - размерность хромосомы
4  F - параметр алгоритма, вещественное положительное число
5  f(x) - фитнесс-функция
6  i,j,k - индексы
7  x,y,v,v1,v2,v3 - хромосомы
8
9  Begin
10     i := 0
11     Init population P(0)
12
13     While (stop condition not reached) do
14         For each x from P(i) do
15             Choose v1,v2,v3 from P(i) randomly
16             // Mutation:
17             v := v1 + F * (v2 - v3)
18             // Crossover:
19             J := rnd[0,N-1]
20             L := rnd[1,N]
21             For k := 0..N-1 do
22                 y[k] := x[k]
23             od
24             For k := 0..L do
25                 y[(J+k) mod N] := v[(J+k) mod N]
26             od
27             // Selection:
28             If (f(x) better than f(y)) then
29                 z := x
30             Else
31                 z := y
32             fi
33             Add z to P(i+1)
34         od
35         i := i+1
36     od
37
38     Get best of population P(i)
39 End
```

Рисунок 3. Блок-схема и псевдокод последовательного алгоритма дифференциальной эволюции

1.5 Оператор мутации для алгоритма дифференциальной эволюции

В алгоритме дифференциальной эволюции в качестве оператора кроссинговера может подходить любой оператор кроссинговера эволюционных алгоритмов, но предпочтительнее использовать операторы, придуманные для непрерывной оптимизации, так как алгоритм рассчитан на непрерывную оптимизацию.

Более подробно стоит рассмотреть операторы мутации для алгоритма дифференциальной эволюции. Все они основаны на так называемых *базовых векторах* v_0 и *дифференциальных векторах* $(v_1 - v_2)$ (которым данный алгоритм обязан своим названием), где v_0, v_1, v_2 - хромосомы, выбранные из текущего поколения. Существует общее шаблонное название алгоритма: DE/x/y/z, где x - способ выбора базовых векторов и построения дифференциальных векторов, y - количество дифференциальных векторов, z - вид оператора кроссовера, который иногда опускается. Из данного названия можно определить оператор мутации с помощью части x и y. В общем виде оператор мутации для алгоритма дифференциальной эволюции выглядит следующим образом:

$$v = v_0 + \sum_i F_i * (v_{1\,i} - v_{2\,i})$$

Ниже приведены примеры операторов мутации:

- *DE/rand/1*. Происходит случайная выборка трех хромосом: v_0, v_1, v_2 и мутантный вектор строится следующим образом: $v = v_0 + F * (v_1 - v_2)$, где F - сила мутации (вещественная положительная константа).
- *DE/rand/2*. Происходит случайная выборка пяти хромосом: v_0, v_1, v_2, v_3, v_4 и мутантный вектор строится следующим образом: $v = v = v_0 + F_1 * (v_1 - v_2) + F_2 * (v_3 - v_4)$, где F_1, F_2 - сила мутации (вещественные положительные константы).
- *DE/best/1*. Происходит случайная выборка двух хромосом: v_1, v_2 и нахождение наиболее оптимальной хромосомы текущего поколения v_{best}, мутантный вектор строится следующим образом: $v = v_{best} + F * (v_1 - v_2)$, где F - сила мутации (вещественная положительная константа).
- *DE/current-to-best/2*. Происходит случайная выборка двух хромосом: v_1, v_2 и нахождение наиболее оптимальной хромосомы текущего поколения v_{best}, а так же используется выбранный на данном шаге вектор из популяции x_i, мутантный вектор строится следующим образом: $v = x_i + F_1 * (v_1 - v_2) + F_2 * (v_{best} - x_i)$, где F_1, F_2 - сила мутации (вещественные положительные константы).

1.6 Обзор существующих модификаций алгоритма

Впервые алгоритм был предложен Шторном и Прайсом в статье "Differential Evolution — A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces" [1], который был подробно описан выше. В дальнейшем алгоритм был модифицирован и исследован в различных статьях. В статье "A Comparative Study of Differential Evolution, Particle Swarm Optimization, and Evolutionary Algorithms on Numerical

Benchmark Problems" [2] было проведено сравнительное исследование алгоритма дифференциальной эволюции, алгоритма роя частиц, генетического алгоритма и сделан вывод о том, что среди этих алгоритмов алгоритм дифференциальной эволюции дает лучшие результаты по точности с ростом размерности задачи. То есть, при размерности равной 30 все из представленных алгоритмов показывали хорошую точность: некоторые из них на некоторых тестах давали более точный результат, чем дифференциальная эволюция, а при размерности равной 100 точность результата, полученного с помощью алгоритма дифференциальной эволюции, уже становилась наилучшей. В статье "Differential Evolution with A New Mutation Operator for Solving High Dimensional Continuous Optimization Problems" [3] была исследована возможность применения нового оператора мутации, являющегося выпуклой суммой двух операторов мутации и использующего самую оптимальную хромосому из всей популяции и самую оптимальную из случайно выбранных:

$$v = \alpha\big(x_i + F * (v_1 - v_2) + F * (v_{best} - x_i)\big) + \beta\big(v_3 + F * (Best - v_4)\big)$$

где $v_1 - v_4$ - хромосомы, выбранные случайным образом, $Best$ - лучшая (с наиболее оптимальной фитнесс-функцией) хромосома на текущей итерации, v_{best} - лучшая хромосома среди $v_1 - v_4$, α, β - коэффициенты выпуклой суммы ($\alpha + \beta = 1$), F - сила мутации, x_i - перебираемый вектор. Исследуется поведение алгоритма с этим оператором мутации на задачах с большой размерностью (100, 500, 1000). В статье "Enhanced parallel Differential Evolution algorithm for problems incomputational systems biology" [4] исследуется параллельная асинхронная островная модель алгоритма и ее применение на некоторых задачах биоинформатики. В статье "RDEL: Restart Differential Evolution algorithm with Local Search Mutation for global numerical optimization" [5] используется новый оператор мутации, использующий идею алгоритма роя частиц и учитывающий значения лучшей и худшей хромосом из популяции:

$$v = v_1 + F_1 * (Best - v_1) + F_2 * (v_1 - Worst)$$

где v_1 - хромосома, выбранная случайным образом, $Best$ - лучшая (с наиболее оптимальной фитнесс-функцией) хромосома на текущей итерации, $Worst$ - худшая хромосома на текущей итерации, F_1, F_2 - сила мутации. Суть данного оператора состоит в том, что случайный вектор передвигается оператором в сторону ближе к лучшему значению и дальше от худшего значения. В этой статье также вводится рестарт-механизм, суть которого заключается следующем: если фитнесс-функция пробного вектора и перебираемого вектора отличаются незначительно ($|f(v) - f(x_i)| < \varepsilon$, для некоторого $\varepsilon > 0$), то оператор мутации запускается заново.

Стоит отметить, что в оригинальном алгоритме, описанном Шторном и Прайсом, использовался специальный DE кроссинговер, в дальнейших исследованиях алгоритма разными авторами всегда использовался однородный кроссинговер. Существует несколько реализаций параллельного алгоритма, которые описаны в различных статьях, но все они базированы на

островной модели, о которой будет рассказано ниже. Помимо вышеуказанных работ алгоритм дифференциальной эволюции исследовался еще в нескольких работах, но в них не было внесено ничего нового, кроме новых операторов мутации, но это незначительная модификация, поэтому рассмотрение этих работ в данном обзоре не велось.

1.7 Актуальность алгоритма дифференциальной эволюции

Алгоритм используется в задачах во многих прикладных областях, так как его можно отнести к области искусственного интеллекта и машинного обучения и распознавания, а данная технология, как известно, применима во многих прикладных науках. Вот некоторые примеры задач, для решения которых используется алгоритм дифференциальной эволюции:

- Задачи оптимального портфельного инвестирования
- Оптимизация параметров аэрокосмических систем
- Расчет параметров микроскопического оптического потенциала упругого рассеяния π+, на ядрах
- Используется в поисковой системе Яндекс для улучшения алгоритмов ранжирования
- Геологическая разведка полезных ископаемых

2. Постановка задачи

Задачи в этой работе состояли в следующем:

- Разработать параллельный алгоритм дифференциальной эволюции, учитывающий возможность настройки параметров алгоритма, вычислительной системы и целевой функции
- Реализовать предложенный параллельный алгоритм для систем с распределенной памятью с использованием технологии MPI
- Определить набор тестовых функций и параметров
- Провести исследование разработанного параллельного алгоритма на выбранном наборе тестовых функций

3. Построение и исследование решения задачи

Прежде всего стоит отметить, что алгоритм дифференциальной эволюции был разработан для задач непрерывной оптимизации, и действительно, параметры мутации - константы F_i выбираются из непрерывного поля действительных чисел, а операции в операторе мутации хорошо подходят лишь для задач непрерывной оптимизации. Безусловно константы F_i можно было бы определить как некоторые операторы,

действующие на хромосому, в результате чего получалась бы хромосома, но подбор подходящего оператора, который сохраняет свойства алгоритма, отнюдь не тривиален.

Поэтому было проведено исследование алгоритма на задаче дискретной оптимизации - задаче о рюкзаке, как показано в таблице ниже:

Алгоритм дифференциальной эволюции		Генетический алгоритм	
Относительная точность[2]	Кол-во итераций	Относительная точность	Кол-во итераций
0,89	656	0,9	120
0,78	105	0,83	61
0,82	287	0,83	42
0,78	264	0,82	38
0,77	393	0,83	142

Таблица 1. Сравнение алгоритма дифференциальной эволюции с генетическим алгоритмом на примере задачи о рюкзаке.

В качестве модельной задачи была взята задача о рюкзаке со случайной генерацией параметров вещей (веса и стоимости) из интервала [1,500] с количеством вещей в размере 250 и максимальным весом рюкзака в 100000. В качестве хромосомы - бинарный вектор размерности, соответствующей количеству всех предметов (250), которые можно положить в рюкзак.

В генетическом алгоритме при переходе на следующую итерацию последовательно были применены оператор однородного кроссовера с двумя потомками со случайным выбором родителей из прошлой популяции, а после - ко всем потомкам оператор мутации со случайной сменой бита с вероятностью 0,04.

В качестве операций +/- для хромосомы в алгоритме DE - битовая операция XOR, в качестве умножения хромосомы на константу - тождественное отображение хромосомы (умножение на 1), и поэтому оператор мутации для

[2]Относительная точность - $1 - |Approx - Exact|/|Exact|$, где $Approx$ - приближенное решение, полученное с помощью одного из алгоритмов, $Exact$ - точное решение, полученное с помощью метода динамического программирования.

пробного вектора v выглядит следующим образом: $v = v_1 \oplus v_2 \oplus v_3$. В качестве оператора кроссовера был взят однородный кроссовер с одним потомком.

Размер популяции в обоих алгоритмах оставался постоянным.

Стоит заметить, что при любых значениях параметров алгоритм дифференциальной эволюции сходится медленнее и почти всегда к менее точному результату, нежели генетический алгоритм. Отсюда можно сделать вывод, что его нецелесообразно применять к задачам дискретной оптимизации, но имеет смысл применять к задачам непрерывной оптимизации.

3.1 Параллельные модели алгоритма

В качестве основы для одной из реализаций алгоритма была использована классическая для эволюционных алгоритмов островная модель. Суть ее заключается в следующем:

- На каждом из вычислительных узлов генерируется одна или несколько популяций (возможно со своими параметрами), каждая из этих популяций и называется *островом*.
- На каждом острове происходит прорабатывает индивидуальная версия алгоритма
- После каждой итерации происходит синхронизация и проверка условия останова.
- После некоторых итераций возможны обмены частью популяции, если они предусмотрены параллельным алгоритмом.
- После выполнения условий останова и завершения работы алгоритма происходит сбор "лучших" особей с островов и среди них выбирается самая лучшая, которая и выдается в качестве ответа.

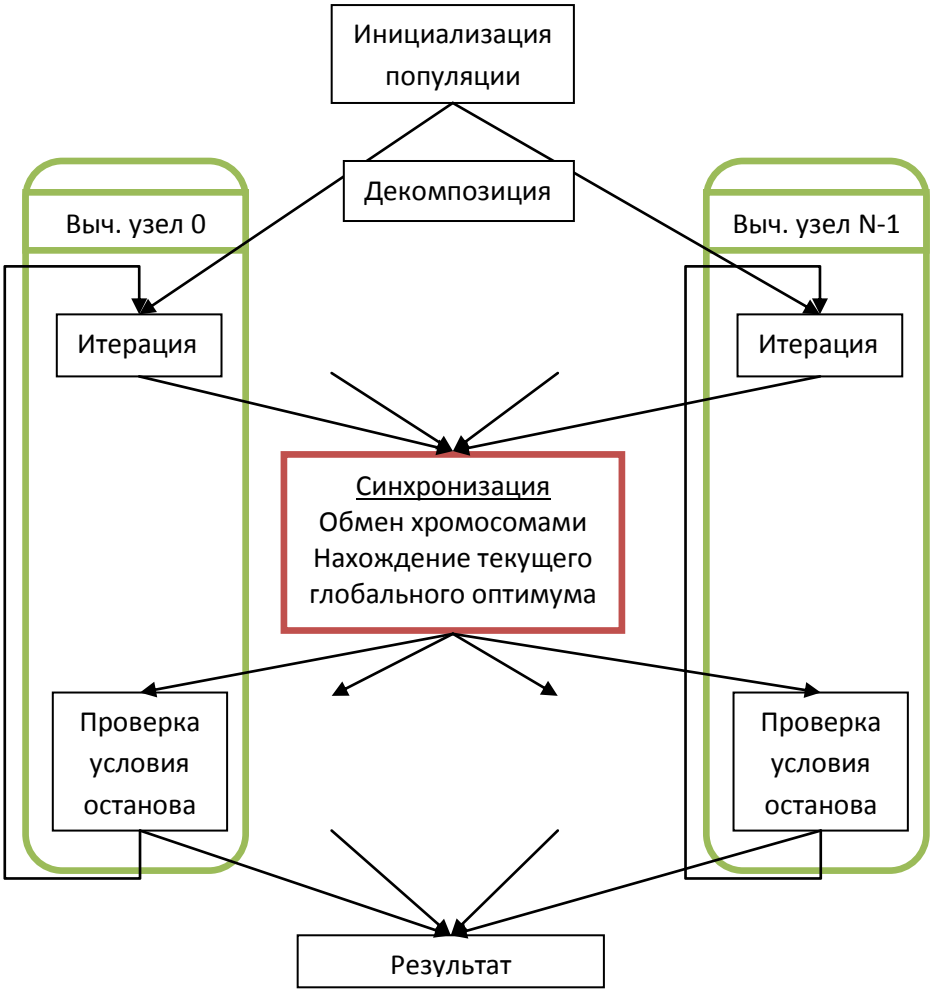


Рисунок 4. Схема островной модели алгоритма дифференциальной эволюции

На рисунке выше показана схема алгоритма, использующего островную модель, из которой можно видеть, что в алгоритме каждая итерация на каждом острове проходит независимо, а синхронизация островов происходит при обмене особями и нахождении глобального оптимума.

Данная модель по сути позволяет искусственно увеличивать популяцию за счет памяти на других узлах, а обмены между островами наравне с операторами мутации не позволяют островам "сваливаться" в локальные экстремумы. Данная модель достаточно хороша: она интуитивно понятна, она универсальна: подходит для любого эволюционного алгоритма, она хорошо изучена, у нее есть несколько параметров для точной настройки (например: количество хромосом для обмена, частота обменов, стратегия обменов).

Но данная модель не дает гарантированного ускорения с ростом количества вычислительных узлов, поэтому была разработана модель, осуществляющая декомпозицию задачи по группам генов. Суть этой модели состоит в следующем:

- Каждая хромосома разделяется на группы генов, каждая из которых ассоциируется с вычислительным узлом.
- Параллельно происходит итерация алгоритма: сначала гены вычисляются с помощью оператора мутации, после - с помощью оператора кроссинговера.
- Происходит синхронизация: вычисляется фитнесс-функция пробного вектора, в зависимости от оптимальности значения фитнесс-функции в следующую популяцию отбирается либо пробный вектор либо перебираемый вектор. Также, запоминается соответствующая фитнесс-функция.
- Выполняется проверка условия останова, и в зависимости от результата, алгоритм либо продолжает свою работу, либо останавливается.

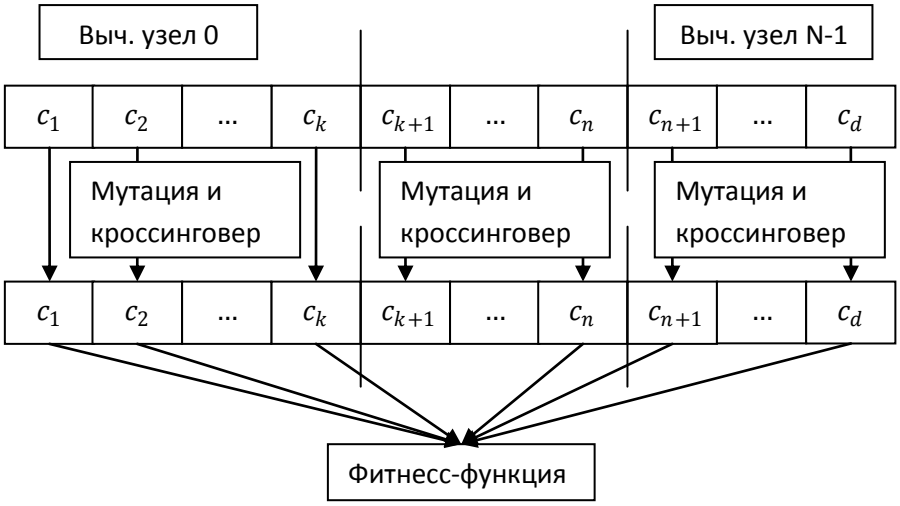


Рисунок 5. Схема итерации в модели декомпозиции по группам генов алгоритма дифференциальной эволюции

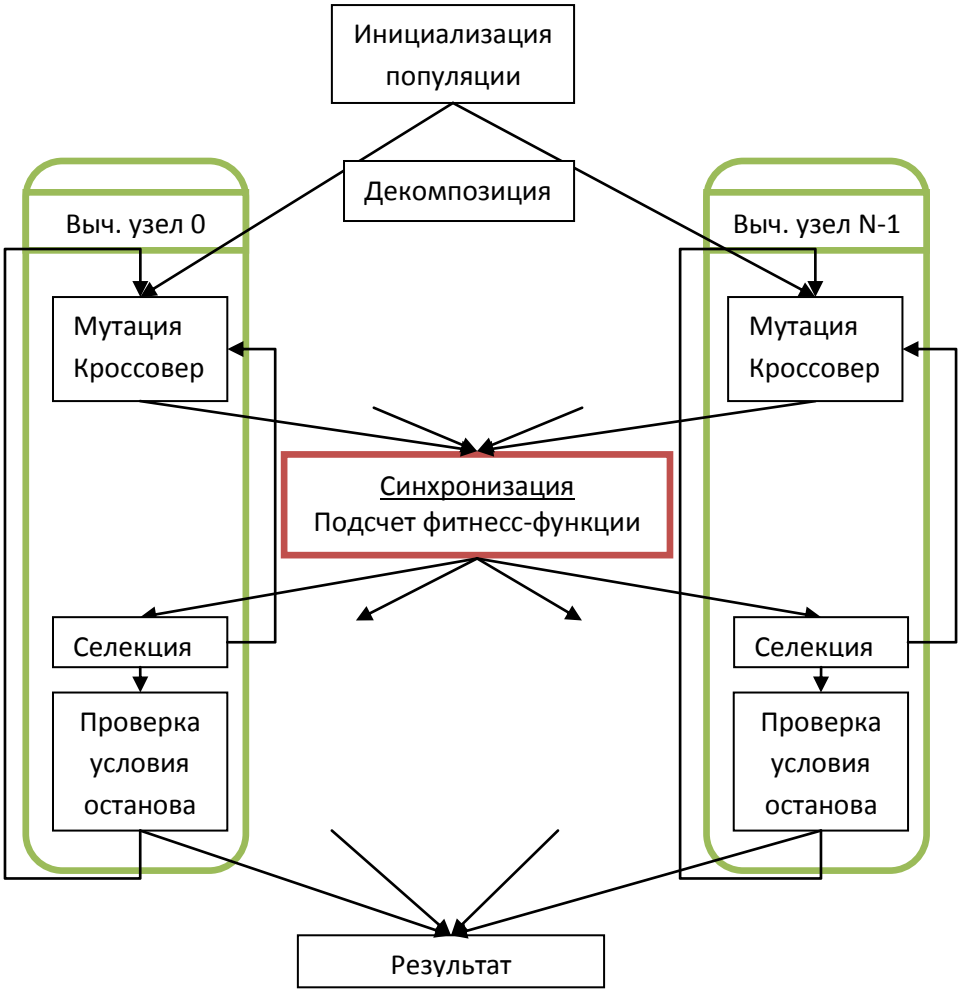


Рисунок 6. Схема модели декомпозиции по группам генов алгоритма дифференциальной эволюции

На рисунках выше показаны схемы итерации алгоритма и всего алгоритма, использующего модель декомпозиции по группам генов для случая параллельных операторов кроссовера, мутации и селекции. На схемах видно, что синхронизация в алгоритме происходит при подсчете фитнесс-функции. Мотивация использования данной модели состоит в том, что каждый ген в алгоритме для некоторых операторов кроссовера вычисляется независимо от других генов, за исключением вычисления фитнесс-функции.

Декомпозиция в представленных выше моделях наглядно показана на рисунке ниже:

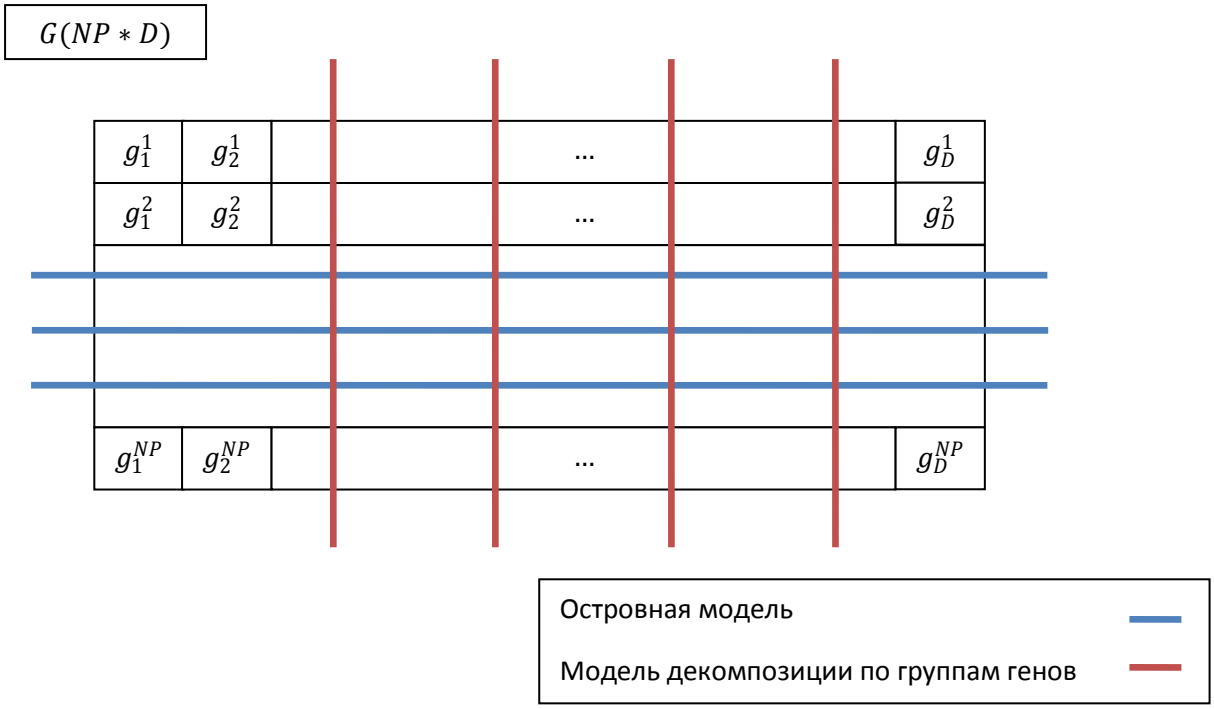


Рисунок 7 Схема моделей декомпозиции

Если представить популяцию в виде матрица G размером $NP * D$, где NP - исходный размер популяции (количество особей), а D - количество генов у одной особи, то декомпозицию по островам и по группам генов можно представить как группировку строк и столбцов матрицы соответственно. Отсюда получаются естественные ограничения на модели: популяцию всегда можно представить в виде множества особей (строк матрицы), поэтому островная модель применима всегда, но не всегда каждую особь можно представить в виде множества генов (столбцов матрицы), поэтому модель декомпозиции по группам генов применима только в случае , если хромосома представима в виде множества генов. При этом, если особь представима в виде генов, модели можно использовать совместно. Ниже будет проведено исследование этих двух моделей параллельного алгоритма.

3.2 Программная реализация и техническая основа

Программная реализация алгоритма выполнена на языке C++ с использованием технологии MPI для параллельной реализации. Программная реализация состоит из следующих частей:

1. Класс с набором параметров
2. Абстрактный интерфейс Chromosome с конкретными реализациями для конкретных задач

3. Абстрактный интерфейс Genetic_operator необходимый для реализации операторов мутации, кроссовера и селекции
4. Три реализации алгоритма: последовательная и две параллельных
5. Различные вспомогательные модули программы
6. Вспомогательные скрипты:
    - Makefile
    - Скрипты для усреднения данных
    - Скрипты для визуализации данных

Программа написана согласно парадигме объектно-ориентированного программирования и так как в программе присутствуют абстрактные интерфейсы, был использован фабричный шаблон проектирования.

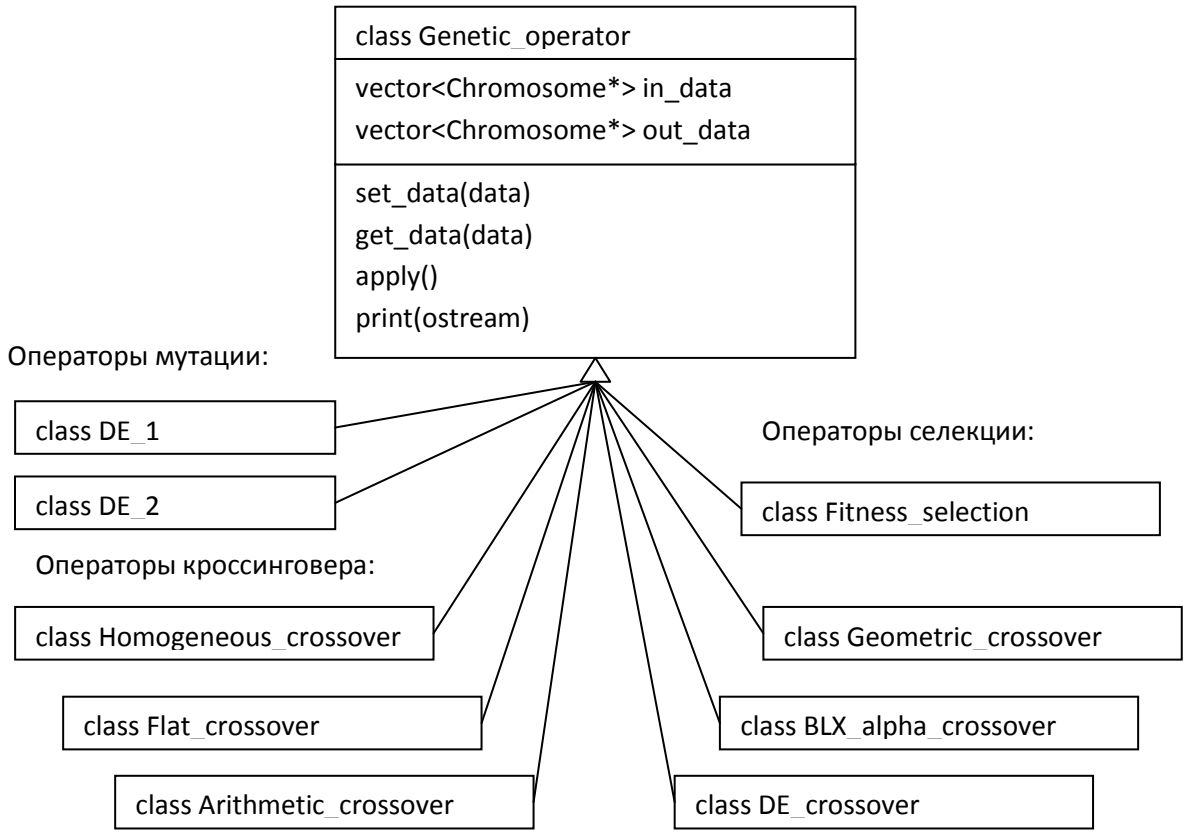


Рисунок 8 Диаграмма интерфейса генетического оператора с реализациями

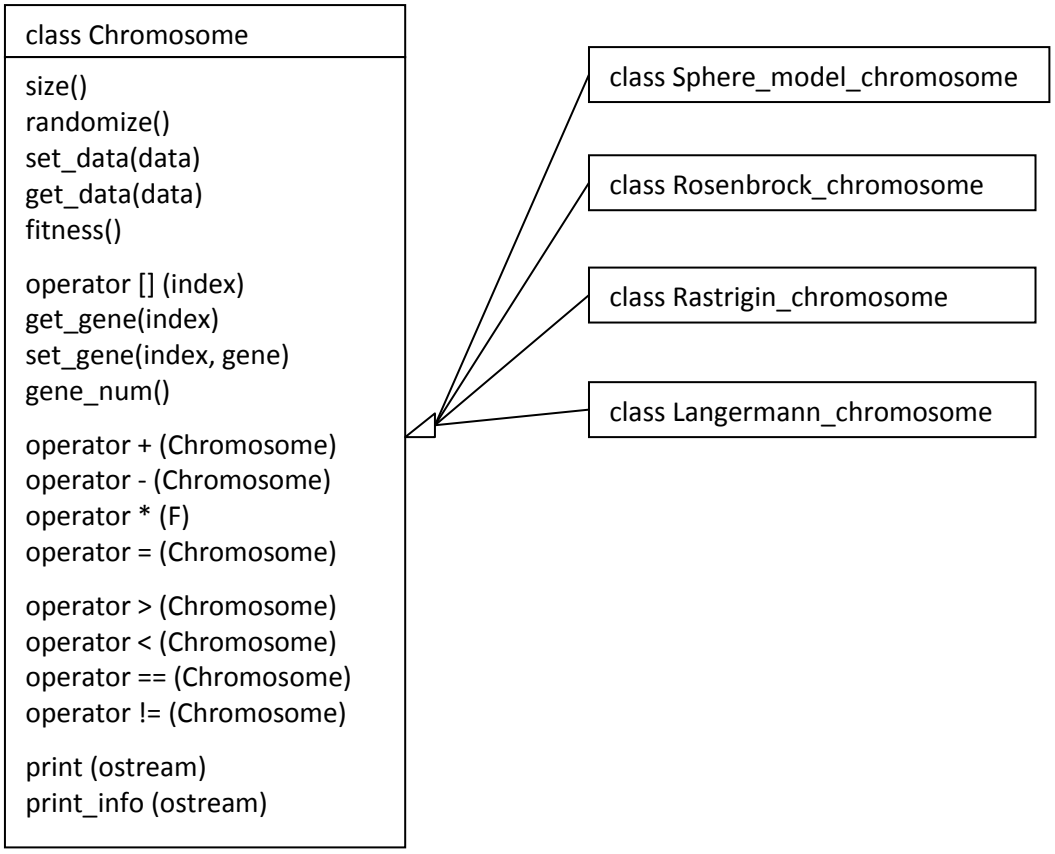


Рисунок 9 Диаграмма интерфейса хромосомы с реализациями

Выше показаны два основных интерфейса в программе (стрелки означают наследование, в блоке сверху указано название класса, ниже - данные и методы). Эти интерфейсы предоставляют основные инструменты для реализации различных операторов и хромосом для различных задач соответственно. Класс 'Genetic_operator' работает по схеме: прием множества хромосом (set_data) - применение оператора (apply) - получение результата (get_data). Класс 'Chromosome' предоставляет функционал для работы с содержимым (данными), которое специфицируется при наследовании, и предоставляет сервис для алгоритма (операции сравнения, +, -, *). Остальные элементы являются служебными и их освещение является излишним. Стоит отметить, что вышеуказанная архитектура была специально разработана, чтобы ее можно было использовать в других эволюционных алгоритмах, например, в генетическом, и используя данную архитектуру, можно строить общую систему эволюционных алгоритмов.

Тесты проведены на суперкомпьютере BlueGene/P: 2048 вычислительных узла с процессором PowerPC 450 и 2 ГБ ОЗУ на узел, подробнее о данной системе можно узнать на странице http://hpc.cmc.msu.ru/bgp.

3.3 Тестовые задачи

Для исследований ниже по умолчанию в качестве условия останова было выбрано достижение оптимального значения функции с точностью до 1e-5 (10^{-5}), и по умолчанию задачи тестировалась на размерности равной 10 и на 32 вычислительных узлах комплекса BlueGene/P. В качестве тестовых функций были выбраны следующие (d - размерность задачи):

1. Модель сферы:

$$f(x) = \sum_{i=1}^{d} x_i^2 \quad x_i \in [-5.12, 5.12]$$

Глобальный минимум этой функции находится в точке $x_{min} = (0, \dots, 0)$ и $f(x_{min}) = 0$. Функция от двух переменных изображена на рисунке ниже:

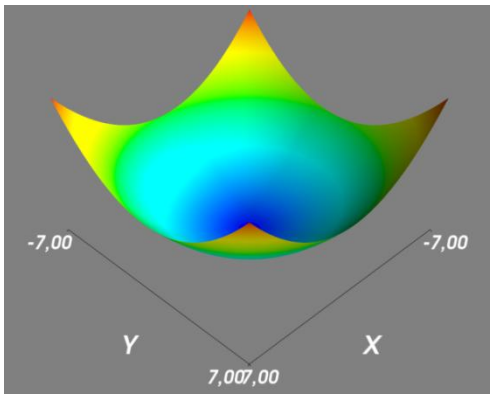


Рисунок 10. Функция "модель сферы" от двух переменных

2. Функции Розенброка:

$$f(x) = \sum_{i=1}^{d-1} \left(100\left(x_{i+1} - x_i^2\right)^2 + (1 - x_i)^2\right) \quad x_i \in [-5.12, 5.12]$$

Глобальный минимум этой функции находится в точке $x_{min} = (1, \dots, 1)$ и $f(x_{min}) = 0$. Данная функция представляет собой "долину" с достаточно большой областью, в которой значение функции близко к 0, к которой многие алгоритмы сходятся достаточно быстро, при том, что абсолютный оптимум достигается лишь в одной точке. Функция от двух переменных изображена на рисунке ниже:

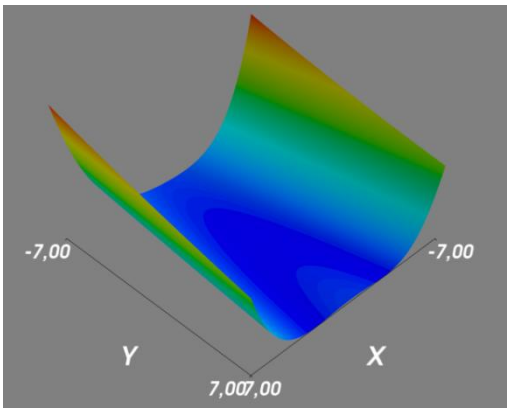


Рисунок 11. Функция Розенброка от двух переменных

3. Функция Растригина:

$$f(x) = 10d + \sum_{i=1}^{d} (x_i^2 - 10 * \cos(2\pi x_i)) \quad x_i \in [-5.12, 5.12]$$

Глобальный минимум этой функции находится в точке $x_{min} = (0, \dots, 0)$ и $f(x_{min}) = 0$. Данная функция является мультимодальной, но при наличии множества локальных минимумов, глобальный минимум только один. Эта функция тестирует алгоритмы на возможность "свалиться" в локальный экстремум. Функция от двух переменных изображена на рисунке ниже:

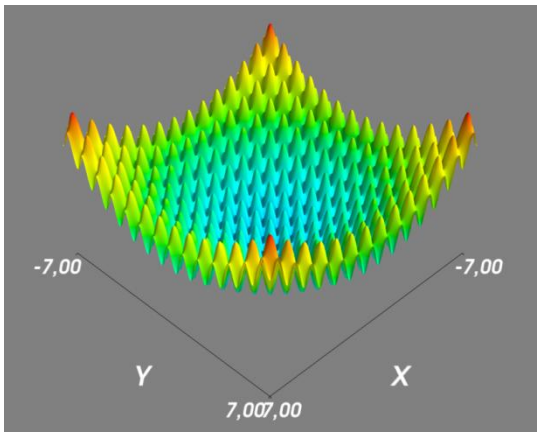


Рисунок 12. Функция Растригина от двух переменных

4. Функция Лангерманна:

$$f(x) = -\sum_{i=1}^{m} c_i \exp\left(-\frac{1}{\pi}\sum_{j=1}^{d}(x_j - A_{ij})^2\right) * \cos\left(\pi \sum_{j=1}^{d}(x_j - A_{ij})^2\right)$$

$$x_i \in [0,10]$$

где $d = 2, m = 5, c = (1,2,5,2,3), A = \begin{pmatrix} 3 & 5 \\ 5 & 2 \\ 2 & 1 \\ 1 & 4 \\ 7 & 9 \end{pmatrix}$

Глобальный минимум находится в точке $x_{min} = (2.00299219, 1.006096)$ и $f(x_{min}) = -5.1621259$. Функция от двух переменных с вышеуказанными параметрами изображена на рисунке ниже:

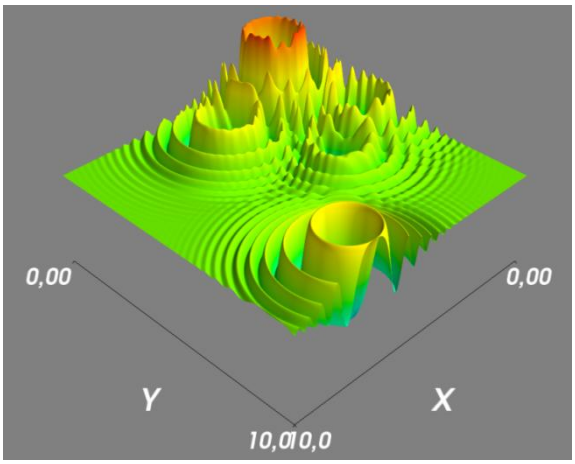


Рисунок 13. Функция Лангерманна от двух переменных

В каждом из исследований ниже показаны средние результаты по 30 запускам.

3.4 Исследование параметров

В этой главе проводилось исследование на двух тестовых функциях: функции Растригина и функции Розенброка, так как эти функции являются типичными представителями задач непрерывной оптимизации.

Алгоритм имеет множество параметров, некоторые из которых характерны для всех вариаций алгоритма, некоторые - только для параллельного алгоритма, некоторые порождаются вариацией других параметров, например вариацией видов генетических операторов. Дерево параметров для островной модели можно увидеть на рисунке ниже, листья дерева - списки конечных параметров:

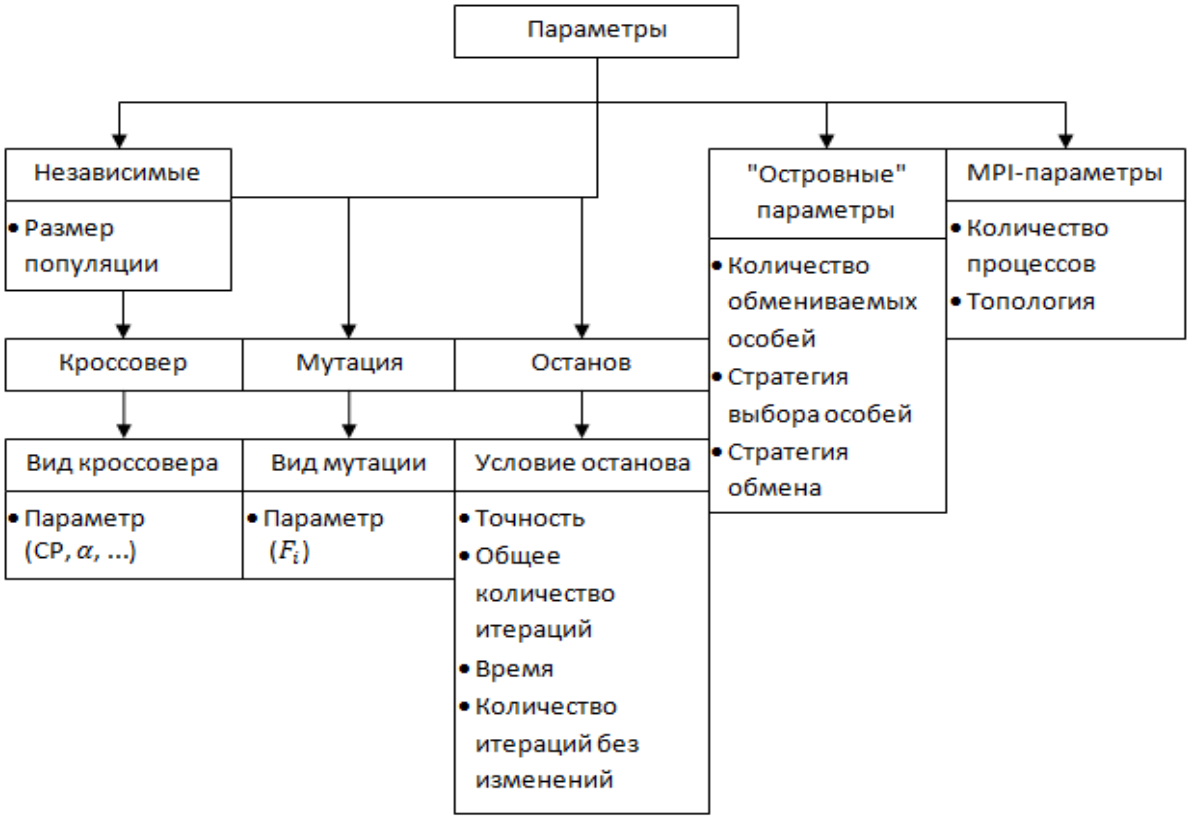
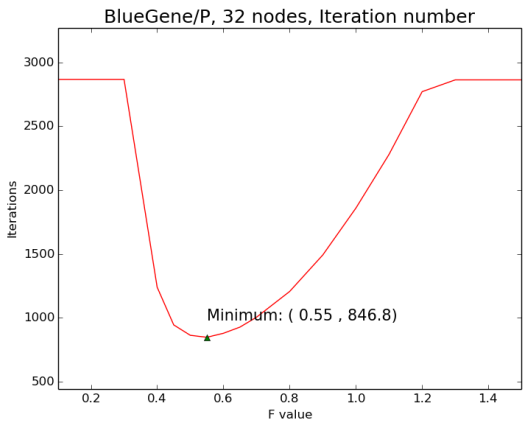


Рисунок 14. Параметры островной модели алгоритма дифференциальной эволюции

Первым для исследования был выбран один из основных параметров алгоритма: константу F - силу мутации. Суть этого параметра состоит в том, что он масштабирует дифференциальные векторы и определяет величину "шага" от базового вектора. Ниже приведены два графика, отражающие сходимость алгоритма в зависимости от величины параметра (слева - количество итераций, за которое алгоритм сошелся к точному решению с заданной точностью, справа - время, за которое алгоритм сошелся к точному решению с заданной точностью):



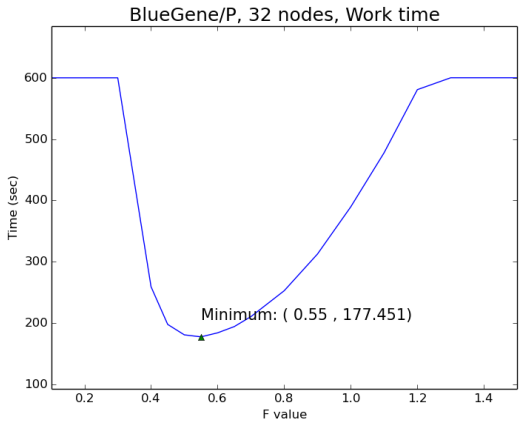


Рисунок 15. Функция Розенброка, зависимость скорости сходимости алгоритма от силы мутации

На графиках выше видно, что оптимальным значением параметра является 0.55, поэтому в дальнейших исследованиях функции Розенброка этот параметр зафиксирован с полученным значением.

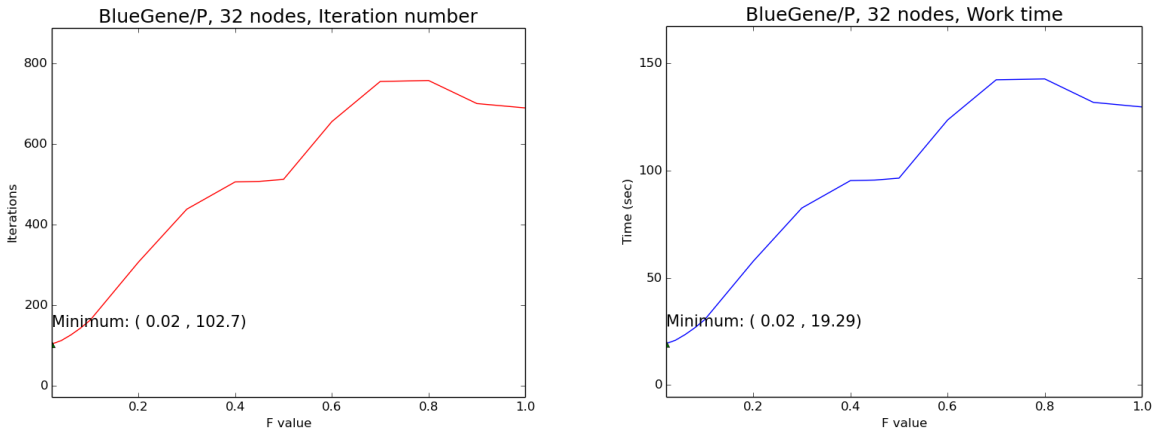


Рисунок 16. Функция Растригина, зависимость скорости сходимости алгоритма от силы мутации

На графиках выше видно, что чем меньше значение силы мутации, тем быстрее сходится алгоритм, но с уменьшением этого значения, падает вероятность того, что алгоритм сойдется к точному решению, то есть, так как функция Растригина мультимодальна, увеличивается вероятность того, что алгоритм сойдется к локальному оптимуму. Поэтому для мультимодальных функций помимо времени работы алгоритма необходимо учитывать вероятность сходимости. Немонотонность графика является следствием немонотонности функции.

Вторым основным параметром для алгоритма является размер популяции, который играет значительную роль в скорости сходимости алгоритма. Популяция, покрывающая все пространство поиска, способна найти оптимум за один шаг; слишком малая популяция способна быстро сойтись к локальному оптимуму; слишком большая популяция будет долго обсчитываться и может не давать прирост в скорости, но прирост во времени подсчета будет давать всегда. Поэтому важно найти оптимальное значение, при котором будет наискорейшим образом достигаться оптимум при минимальных затратах времени на итерацию, и как следствие, на весь алгоритм. Ниже приведены два графика, отражающие сходимость алгоритма в зависимости от размера популяции (слева - количество итераций, за которое алгоритм сошелся к точному решению с заданной точностью, справа

- время, за которое алгоритм сошелся к точному решению с заданной точностью):

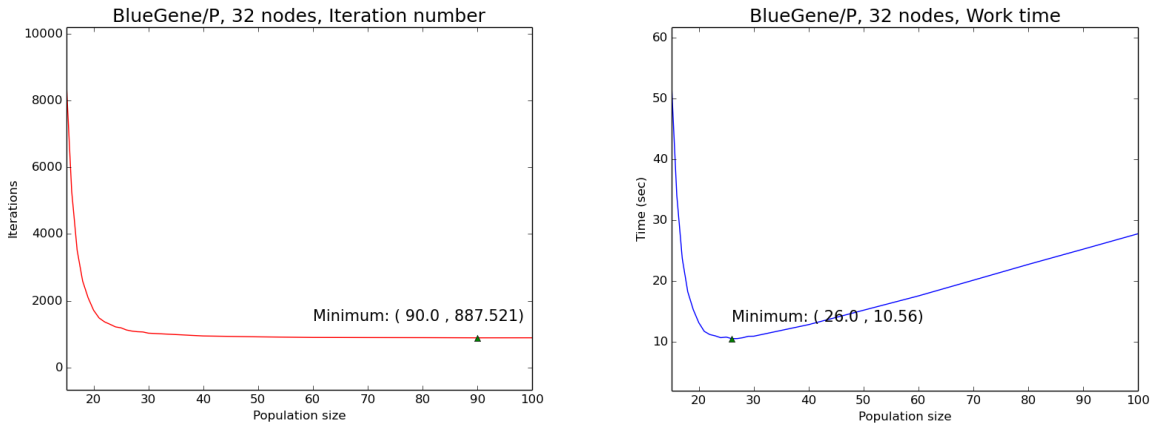


Рисунок 17. Функция Розенброка, зависимость скорости сходимости алгоритма от размера популяции

На левом графике видно, что количество итераций стабилизируется, то есть увеличение популяции не дает существенного выигрыша, но из правого графика можно понять, что из-за роста размера популяции и примерно постоянного количества итераций алгоритма, время работы алгоритма растет линейно относительно размера популяции. Стоит отметить, что минимум по времени достигается на достаточно малом размере популяции - 26 особей, но при этом не стоит забывать, что исследование проводится на 32 вычислительных узлах, и поэтому общее количество особей на всех островах составляет 832.

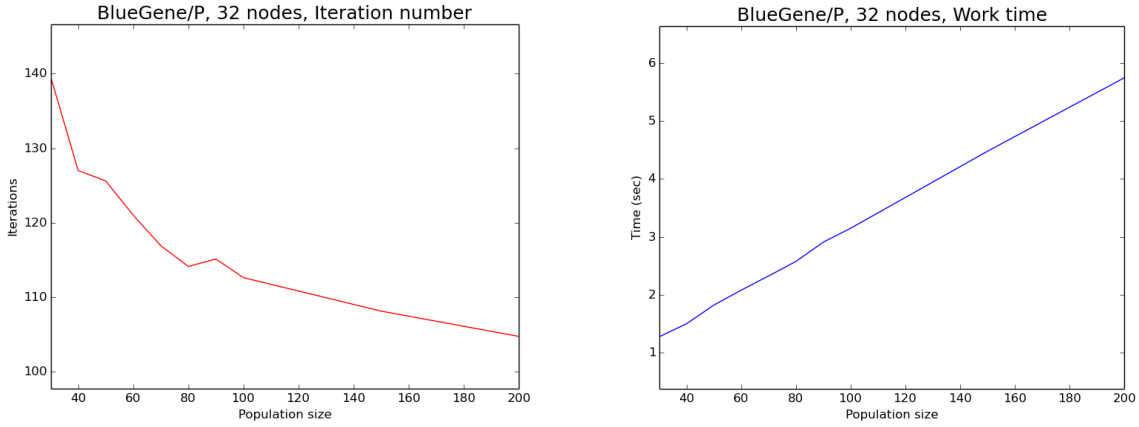


Рисунок 18. Функция Растригина, зависимость скорости сходимости алгоритма от размера популяции

На графиках выше видно, что даже несмотря на уменьшение количества итераций, необходимого для достижения точного решения, время

работы алгоритма растет линейно. Но при этом нельзя однозначно выбирать минимальный размер популяции, так как с уменьшением размера популяции уменьшается вероятность сходимости, как и в случае силы мутации.

Одной из задач этой работы было исследование поведения алгоритма на задачах с большой размерностью, поэтому ниже приведены два графика, отражающие сходимость алгоритма в зависимости от размерности задачи (слева - количество итераций, за которое алгоритм сошелся к точному решению с заданной точностью, справа - время, за которое алгоритм сошелся к точному решению с заданной точностью):

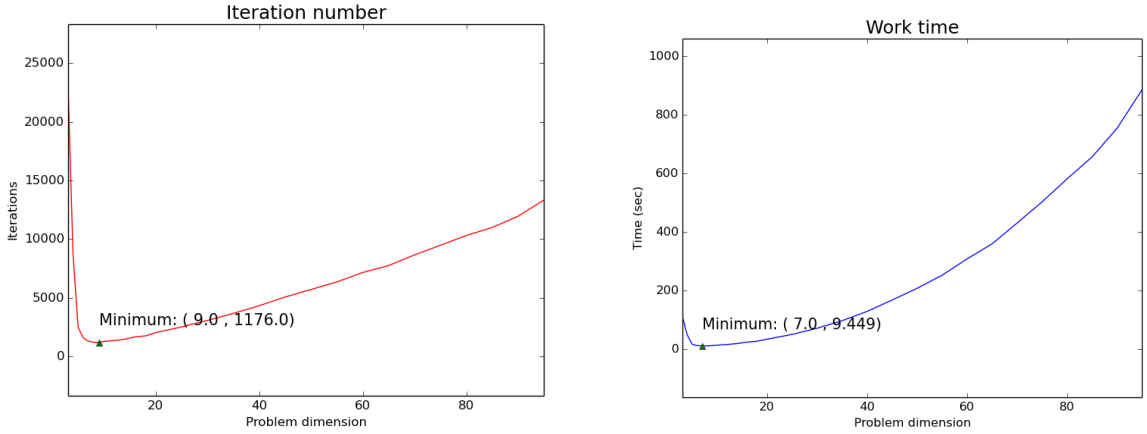


Рисунок 19. Функция Розенброка, зависимость скорости сходимости алгоритма от размерности задачи

Можно видеть, что с ростом размерности растет и количество итераций алгоритма, и так как за одну итерацию с ростом размерности производится больше вычислений, время работы алгоритма растет быстрее количества итераций. Исключением из этого являются малые размерности (до 9), что связано с тем, что оператор кроссовера меняет гены, и чем меньше генов в хромосоме, тем меньше можно поменять, отсюда и меньше генетическое разнообразие и долгая сходимость.

3.5 Исследование островной модели

В этой главе и далее проводилось исследование задач с большой размерностью на тестовой задаче "модель сферы" с размерностью 1024. Островная модель дает следующие существенные преимущества по сравнению с последовательным алгоритмом:

- Увеличение числа особей за счет увеличения суммарного объема памяти узлов

- Возможность запустить алгоритм на каждом из островов с индивидуальными параметрами
- Появление возможности одновременной сходимости сразу к нескольким локальным оптимумам на каждом острове

Но данная модель не масштабируется по времени, более того, с ростом числа вычислительных узлов, растет число обменов, и как следствие, растет общее время обменов, общее время работы алгоритма. Но при этом можно наблюдать некоторое масштабирование точности результата, иногда даже сверхлинейное.

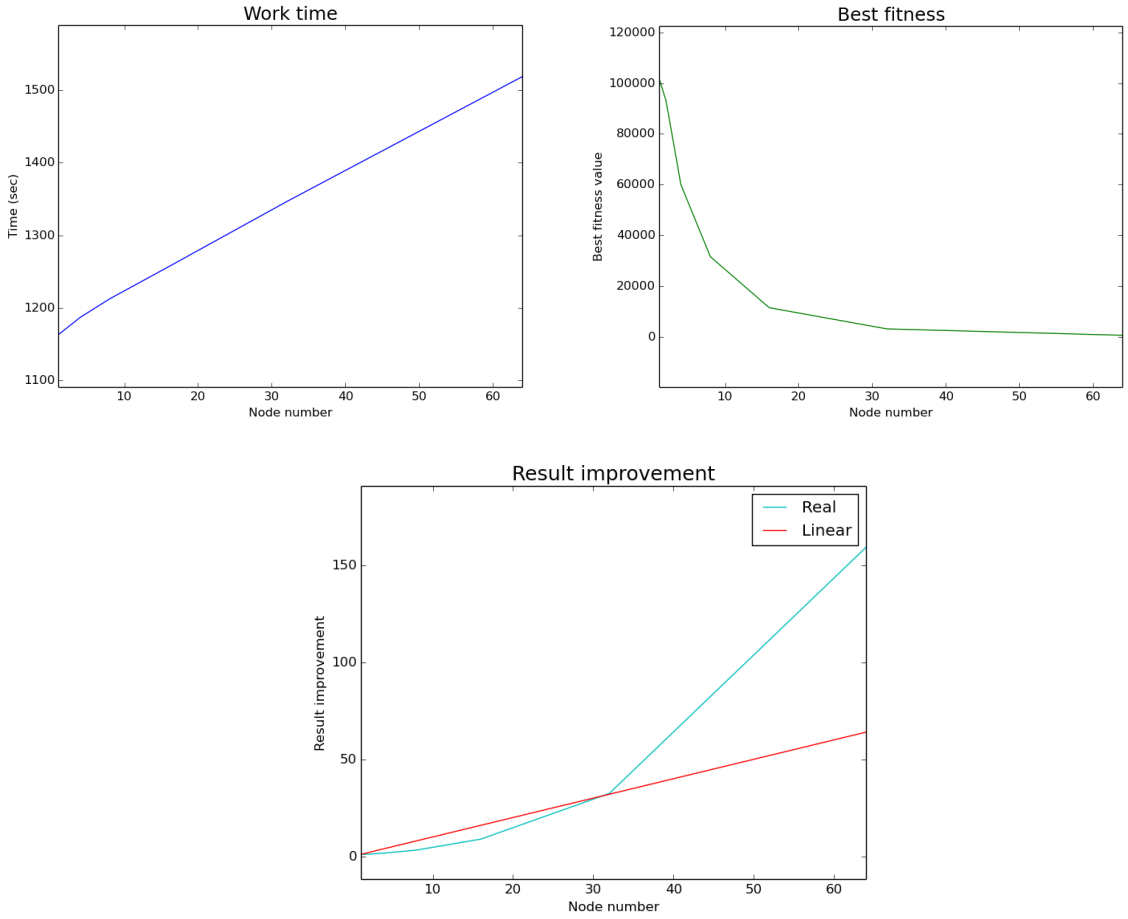


Рисунок 20. Время работы, лучшее значение фитнесс-функции и улучшение точности результата в зависимости от количества вычислительных узлов

В качестве условия останова в данном случае было выбрано достижение алгоритмом 3000 итераций. Последняя из метрик на рисунке - улучшение точности результата: высчитывается аналогично ускорению: $I = R_1/R_p$, где R_1 - результат работы алгоритма на одном вычислительном узле, R_p - результат работы алгоритма на p вычислительных узлах. А сверхлинейное улучшение точности результата вызвано тем, что точность результата зависит от размера популяции не линейно.

3.6 Исследование модели декомпозиции по группам генов

Островная модель может показывать неплохие результаты, но с ростом числа вычислительных узлов, как было показано выше, гарантированно увеличивается время работы и может увеличится точность результата. Но для размера популяции, как было выяснено ранее, существует асимптотический предел, поэтому начиная с некоторого количества островов, точность результата будет увеличиваться незначительно. Также, стоит отметить, что островная модель решает проблему с масштабированием по памяти лишь косвенно: за счет распределения популяции по островам.

Поэтому рассмотрим вторую модель параллельного алгоритма. Время одной итерации последовательного алгоритма состоит из времени работы операторов мутации, кроссовера, селекции, оператор селекции по сути состоит из вычисления фитнесс функции и пренебрежимо малого времени сравнения двух значений фитнесс-функции, поэтому время одной итерации равно:

$$T_{seq} = T_{mutation} + T_{crossover} + T_{fitness}$$

В операторе мутации каждый ген считается независимо, поэтому время одной итерации параллельного алгоритма выглядит следующим образом:

$$T_{par} = \frac{T_{mutation}}{p} + T_{par\ crossover} + T_{par\ fitness}$$

В качестве оператора кроссовера можно выбрать такой оператор, который работает с каждым геном независимо (например, однородный кроссовер), но также, можно использовать синхронную версию оригинального оператора кроссовера, в которой один процесс будет генерировать случайные числа и рассылать их всем с помощью широковещательной рассылки, и все процессы в зависимости от чисел будут работать со своей группой генов независимо, поэтому время работы оператора синхронного кроссовера равно:

$$T_{syn\ crossover} = T_{bcast} + T_{loc\ calc} = O(\log p) + \frac{T_{seq\ calc}}{p}$$

А время работы параллельного оператора кроссовера с независимым подсчетом групп генов равно:

$$T_{par\ crossover} = T_{loc\ calc} = \frac{T_{seq\ calc}}{p}$$

В прикладных задачах могут встретиться фитнесс-функции любого вида, но чаще всего в реальных задачах возникают функции вида $f(\sum_i f_i(x_{j\ 1}, \dots, x_{j\ k}))$ или $f(\prod_i f_i(x_{j\ 1}, \dots, x_{j\ k}))$, для которых гены можно сгруппировать так, чтобы $f_i(x_{j\ 1}, \dots, x_{j\ k})$ можно было бы вычислять независимо на каждом узле, либо по крайней мере с минимальным числом и объемом предварительных пересылок, для таких функций справедливо:

$$T_{par\ fitness} = T_{prepare} + T_{loc\ calc} + T_{reduce} = T_{prepare} + \frac{T_{seq\ calc}}{p} + O(\log p)$$

Подобная оценка распространяется и на другие функции (не только суммирование и взятие произведения), для которых можно сделать редукцию за время порядка логарифма. И очевидно, что это справедливо не для всех функций, например, для функции $f(x_1 + f(x_2 + \cdots f(x_n) \dots))$, где $f(x)$ - нелинейная функция (например, $f(x) = \sin(x)$) данная оценка не будет справедлива. Подобные фитнесс-функции придется считать за $O(d)$, где d - количество генов. Также, утверждение о том, что число и объем предварительных пересылок малы, позволяет нам считать, что $T_{prepare}$ достаточно мало, что, вообще говоря, тоже не всегда верно.

Итого, для вышеуказанных функций мы получаем следующую оценку времени одной итерации параллельного алгоритма:

$$T_{par} = \frac{T_{seq}}{p} + O(\log p)$$

Подобная оценка позволяет утверждать, что алгоритм при больших размерностях достаточно хорошо масштабируется в смысле сильной масштабируемости, что подтверждают эксперименты. В качестве условия останова, как и в исследовании предыдущей модели, было выбрано достижение алгоритмом 3000 итераций.

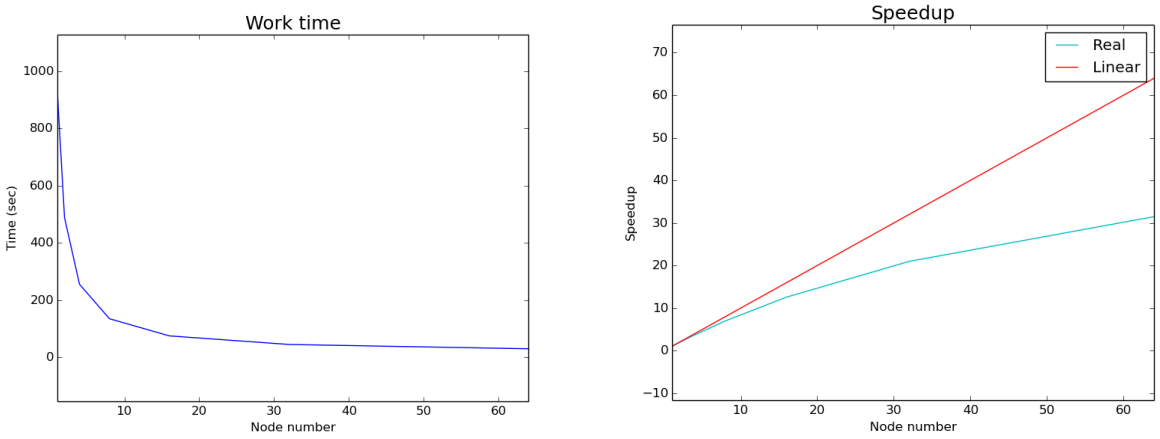


Рисунок 21. Время работы и ускорение алгоритма в зависимости от количества вычислительных узлов

Точность полученного результата оставалась примерно постоянной в каждом эксперименте, что является ожидаемым результатом, так как параметры алгоритма не изменялись - с ростом вычислительных узлов изменялось лишь удельное количество генов на вычислительный узел. Нелинейность ускорения вызвана тем, что с ростом числа вычислительных узлов уменьшается количество генов на узел и добавочная часть обменов между процессами $O(\log p)$ начинает превалировать над временем параллельного подсчета генов $\frac{T_{seq}}{p}$.

4. Заключение

В результате представленной работы было разработано и реализовано две модели параллельного алгоритма дифференциальной эволюции. Выяснено, что островная модель улучшает точность результата с ростом числа вычислительных узлов, а модель декомпозиции по группам генов с ростом числа вычислительных узлов при постоянной точности результата и количестве итераций ускоряется в смысле времени работы. Таким образом, распараллеливание с помощью островной модели хорошо подходит, если пользователь алгоритма стремится к точности результата, а модель декомпозиции по группам генов подходит, если пользователь стремится к скорости работы алгоритма. Реализованные модели можно использовать в качестве основы для генетических алгоритмов благодаря интерфейсным реализациям хромосомы и генетического оператора. Проведено эмпирическое исследование основных параметров алгоритма, которое позволяет отталкиваясь от полученных результатов, подбирать параметры для конкретной задачи.

Подводя итог всему вышесказанному, можно утверждать, что алгоритм дифференциальной эволюции показывает хорошие результаты при оптимизации непрерывных функций непрерывного аргумента, алгоритм сходится к оптимуму мультимодальной функции. Стоит отметить, что имеется пространство для дальнейшей оптимизации алгоритма как и на островной модели, так и на модели декомпозиции по группам генов, в том числе возможно локальное распараллеливание с помощью технологий OpenMP и CUDA.

Литература

[1] **Kenneth Price and Rainer Storn.** *Differential Evolution — A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces, 1995.*

[2] **Jakob Vesterstrom, Rene Thomsen.** *A Comparative Study of Differential Evolution, Particle Swarm Optimization, and Evolutionary Algorithms on Numerical Benchmark Problems*

[3] **Xuemei You.** *Differential Evolution with A New Mutation Operator for Solving High Dimensional Continuous Optimization Problems, 2010.*

[4] **D.R. Penas, J.R. Banga, P. González, R. Doallo.** *Enhanced parallel Differential Evolution algorithm for problems incomputational systems biology, 2015.*

[5] **Ali Wagdy Mohamed.** *RDEL: Restart Differential Evolution algorithm with Local Search Mutation for global numerical optimization, 2014.*

[6] **Marcin Molga, Czesław Smutnicki.** *Test functions for optimization needs, 2005.*

[7] **Т.В. Панченко.** *Генетические алгоритмы, 2007.*

[8] **Andries P. Engelbrecht.** *Computational Intelligence, pp. 237 - 260, 2007.*

[8] Тестовые функции *http://www.sfu.ca/~ssurjano* (Май, 2016)

[9] Тестовые функции *http://infinity77.net/global_optimization/test_functions_nd_L.html* (Май, 2016)

[10] Эволюционные алгоритмы *https://ru.wikipedia.org/wiki/Evolutionary_algorithm* (Май, 2016)

[11] Алгоритм дифференциальной эволюции *https://ru.wikipedia.org/wiki/Differential_evolution* (Май, 2016)