Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

Лабораторная работа № 16

по курсу «Методы оптимизации»

«Параллельный генетический алгоритм»

Студент группы ИУ9-81Б Окутин Д.А.

Преподаватель Посевин Д. П.

Москва 2025

1 Задание

Необходимо реализовать параллельный генетический алгоритм (миграцию). Рабочие и Хозяин должны находиться на localhost. Обмен данными между Рабочими и Хозяином должны происходить по протоколу WebSockets. Время изоляции должно быть параметром. Количество Рабочих также должно меняться. Необходимо сравнить результаты работы классического алгоритма и миграций. Необходимо сравнить время в секундах, и качественно характер формирования особей.

2 Реализация

Алгоритм работы

1. **Инициализация**:
   - Мастер запускается на $\mathrm{localhost}$ и ожидает подключений рабочих.
   - Рабочие генерируют начальную популяцию и вычисляют приспособленность.
2. **Эволюция**:
   (a) **Селекция и размножение**: Рабочие создают новое поколение через кроссинговер и мутацию.
   (b) **Миграция**: Каждые $\mathrm{ISOLATION\_TIME}$ поколений мастер собирает лучших особей от всех рабочих и рассылает их остальным.
   (c) **Интеграция мигрантов**: Рабочие заменяют часть своей популяции на полученных мигрантов.
3. **Критерий остановки**: Алгоритм завершается при достижении значения приспособленности или максимального числа поколений.

Визуализация

- **Данные**: История поколений хранится в history.json, включая гены особей и идентификаторы рабочих.
- **2D-график**: Контурный график функции Розенброка с наложенными точками популяции. Цвета соответствуют разным рабочим.
- **3D-график**: Поверхность функции с отметками значений f(x,y) для каждой особи.
- **Анимация**: Показывает динамику перемещения особей к минимуму функции. Кадры генерируются для каждого поколения.

Параметры

- POPULATION_SIZE: Размер популяции (по умолчанию 100).
- MAX_GENERATIONS: Максимальное число поколений (200).
- ISOLATION_TIME: Период миграции (20 поколений).
- CROSSOVER_PROB: Вероятность кроссинговера (0.9).
- MUTATION_PROB: Базовая вероятность мутации (0.1).

Исходный код программмы представлен в листинге 1 - 5.
Описание листинга 1.

- **Структура** Individual: Содержит гены (вектор вещественных чисел) и значение приспособленности (fitness).
- evaluate!: Вычисляет приспособленность на основе функции Розенброка

  Суммируется для всех пар генов, результат сохраняется с отрицательным знаком (минимизация).
- crossover!: Одноточечный кроссинговер. Точка разрыва (pt) выбирается случайно.

- *mutate*!: Добавляет шум из нормального распределения (с вероятностью p_mut).

Листинг 1: common.jl

```
module Common

export Individual, evaluate!, crossover!, mutate!, select_parents!

mutable struct Individual
    genes::Vector{Float64}
    fitness::Float64
end


function evaluate!(ind::Individual)
    val = 0.0
    for i in 1:length(ind.genes)-1
        xi, xj = ind.genes[i], ind.genes[i+1]
        val += 100 * (xj - xi^2)^2 + (1 - xi)^2
    end
    ind.fitness = -val
end

function crossover!(p1::Individual, p2::Individual)
    pt = rand(1:length(p1.genes))
    c1 = Individual(vcat(p1.genes[1:pt], p2.genes[pt+1:end]), 0.0)
    c2 = Individual(vcat(p2.genes[1:pt], p1.genes[pt+1:end]), 0.0)
    return c1, c2
end

function mutate!(ind::Individual;    =0.1, p_mut=0.1)
    for i in eachindex(ind.genes)
        if rand() < p_mut
            ind.genes[i] +=     * randn()
        end
    end
end

function select_parents!(pop::Vector{Individual}; k=3)

    return sort(pop, by = x -> x.fitness, rev = true)[1:k]
end

end
```

Описание листинга 2.

- **Инициализация популяции**: Гены генерируются в диапазоне [-5,5] с шагом 0.2.
- breed_next: Создает новое поколение:
    1. Нормализация генов для расчета расстояния между родителями.
    2. Вероятность мутации зависит от расстояния: pmut = 1 - dist.
- **Миграция**: Отправка состояния мастеру через WebSockets и получение мигрантов.
- **Интеграция мигрантов**: Замена худших особей в текущей популяции на полученных мигрантов.

Листинг 2: worker.jl

```
include("common.jl")
using .Common, HTTP, HTTP.WebSockets, JSON3, Random

#



const HOST             = get(ENV, "HOST", "127.0.0.1")
const PORT             = get(ENV, "PORT", "8084")
const SERVER_URL       = "ws://$HOST:$PORT"
const POPULATION_SIZE  = parse(Int, get(ENV, "POP_SIZE", "100"))
const MAX_GENERATIONS  = 200
const CROSSOVER_PROB   = 0.9
const MUTATION_PROB    = 0.1
const GENE_COUNT       = 2
const STOP_FITNESS     = 1e-4
const N                = 3
#




function init_population(n::Int, gene_len::Int)
    pop = [Individual(rand(-5.0:0.2:5.0, gene_len), 0.0) for _ in 1:n]
    for ind in pop
        evaluate!(ind)
    end
    return pop
end

```

```
25 function breed_next(pop::Vector{Individual})
26     gene_len = length(pop[1].genes)
27     D = gene_len
28 
29     maxi = [ maximum(ind.genes[j] for ind in pop) for j in 1:gene_len ]
30     mini = [ minimum(ind.genes[j] for ind in pop) for j in 1:gene_len ]
31 
32     children = Individual[]
33     while length(children) < POPULATION_SIZE
34         mom, dad = select_parents!(pop)
35 
36         if rand() < CROSSOVER_PROB
37             c1, c2 = crossover!(mom, dad)
38         else
39             c1, c2 = deepcopy(mom), deepcopy(dad)
40         end
41 
42         norm_diff_sum = 0.0
43         for i in 1:gene_len
44             denom = maxi[i] - mini[i]
45             if denom != 0.0
46                 norm = abs((mom.genes[i] - dad.genes[i]) / denom)
47                 norm_diff_sum += norm^D
48             end
49         end
50 
51         dist = norm_diff_sum / N
52         mutate!(c1, p_mut = 1 - dist)
53         mutate!(c2, p_mut = 1 - dist)
54         evaluate!(c1)
55         evaluate!(c2)
56         push!(children, c1, c2)
57     end
58 
59     return children[1:POPULATION_SIZE]
60 end
61 
62 function send_state(ws, generation::Int, pop::Vector{Individual})
63     payload = Dict(
64         "gen" => generation,
65         "pop" => [JSON3.write(ind) for ind in pop]
66     )
67     HTTP.WebSockets.send(ws, JSON3.write(payload))
68 end
69 
70 function receive_migrants(ws)
```

```
71      raw = HTTP.WebSockets.receive(ws)
72      doc = JSON3.read(String(raw))
73      migrants = [JSON3.read(String(s), Individual) for s in doc["migrants
        "]]
74      return migrants
75  end
76
77  function integrate_migrants(pop::Vector{Individual}, m::Vector{
        Individual})
78      if isempty(m)
79          return pop
80      end
81      sorted = sort(pop, by = x->x.fitness)
82      for (i, mig) in enumerate(m)
83          sorted[i] = mig
84      end
85      return sorted
86  end
87
88  function evolve!(ws)
89      population = init_population(POPULATION_SIZE, GENE_COUNT)
90      for gen in 1:MAX_GENERATIONS
91          population = breed_next(population)
92          send_state(ws, gen, population)
93
94          migrants = receive_migrants(ws)
95          population = integrate_migrants(population, migrants)
96
97          best_val = -maximum(ind->ind.fitness, population)
98          println("Best generation $gen -> $(round(best_val, digits=4))")
99          if best_val < STOP_FITNESS
100             println("Last generation $gen: fitness < $STOP_FITNESS")
101             break
102         end
103     end
104
105     final_best = -maximum(ind->ind.fitness, population)
106     println("Best fitness -> $(round(final_best, digits=4))")
107 end
108
109 function main()
110     println("[Worker] connecting $SERVER_URL    ")
111     HTTP.WebSockets.open(SERVER_URL) do ws
112         println("[Worker] connected")
113         evolve!(ws)
114     end
```

```
115 end
116
117 main()
```

Описание листинга 3.

- Управляет подключением рабочих через WebSockets.
- Собирает данные о популяциях и поколениях.
- Каждые ISOLATION_TIME поколений отправляет лучших особей от всех рабочих в качестве мигрантов.
- Сохраняет историю эволюции в файл history.json.

Листинг 3: master.jl

```
1  include("common.jl")
2  using .Common, HTTP, HTTP.WebSockets, JSON3, Dates, Statistics
3
4  const ISOLATION_TIME = parse(Int, get(ENV, "ISO_TIME", "30"))
5  const HOST           = get(ENV, "HOST", "127.0.0.1")
6  const PORT           = parse(Int, get(ENV, "PORT", "8084"))
7
8  populations = Dict{String, Vector{Individual}}()
9  gen_counts  = Dict{String, Int}()
10 default_time = rand(0.5:0.001:0.6, 1)
11
12 const COUNTER = Ref(0)
13 next_uid() = (COUNTER[] += 1; "worker" * string(COUNTER[]))
14
15 # ===                                                    ===
16 history = []
17
18 HTTP.WebSockets.listen(HOST, PORT) do ws
19     uid = next_uid()
20     println(" $uid connected")
21     populations[uid] = []
22     gen_counts[uid]  = 0
23
24     try
25         t_start = time()
26
27         for raw in ws
28             data       = String(raw)
29             payload    = JSON3.read(data)
```

```
30              gen         = payload["gen"]
31              pop_serial = payload["pop"]
32              pop         = [JSON3.read(s, Individual) for s in pop_serial]
33
34              populations[uid] = pop
35              gen_counts[uid]  = gen
36
37              # ==                                        ==
38              push!(history, Dict(
39                  "gen" => gen,
40                  "worker" => uid,
41                  "points" => [ind.genes for ind in pop]
42              ))
43
44              # ==                                        ==
45              fits     = [ind.fitness for ind in pop]
46              best_val = -maximum(fits)
47              avg_val  = -mean(fits)
48
49              println("Generation $gen;  $uid -> best = $(round(best_val,
    digits=4)), avg = $(round(avg_val, digits=4))")
50
51              # ==                                        ==
52              migrants = Individual[]
53              if gen % ISOLATION_TIME == 0
54                  for p in values(populations)
55                      if !isempty(p)
56                          _, idx = findmax([ind.fitness for ind in p])
57                          push!(migrants, p[idx])
58                      end
59                  end
60              end
61
62              msg_out = JSON3.write(Dict("migrants" => [JSON3.write(m) for
    m in migrants]))
63              HTTP.WebSockets.send(ws, msg_out)
64          end
65
66          println("PARALLEL TIME:", time() - t_start)
67          println("DEFAULT TIME: ", default_time)
68
69      catch e
70          @warn " error   $uid: $e"
71
72      finally
73          println("  $uid disconnected")
```

```
74          delete!(populations, uid)
75          delete!(gen_counts, uid)
76
77          # ===                                          ===
78          open("history.json", "w") do f
79              JSON3.write(f, history)
80          end
81          println("[Master] History written to history.json")
82      end
83  end
```

Описание листинга 4.

- Строит 2D-контурный график и 3D-поверхность функции Розенброка.
- Анимация:
  - Для каждого поколения отображаются точки популяции рабочих (разные цвета для каждого рабочего).
  - 2D: Контурный график с точками.
  - 3D: Поверхность с отметками значений функции для особей.
- Результаты сохраняются в GIF-файлы (evolution_2d.gif, evolution_3d.gif).

Листинг 4: vizualize.jl

```
1  using JSON3, Plots
2
3  #
4  history_data = JSON3.read(read("history.json", String))
5
6  #                                                  worker'
7  worker_ids = unique([entry["worker"] for entry in history_data])
8  n_workers = length(worker_ids)
9  colors = palette(:tab10, n_workers)
10 worker_colors = Dict(worker_ids[i] => colors[i] for i in 1:n_workers)
11
12 #
13 grouped_history = Dict{Int, Vector{NamedTuple}}()
14 for entry in history_data
15     gen = entry["gen"]
16     if !haskey(grouped_history, gen)
17         grouped_history[gen] = NamedTuple[]
18     end
```

```
    push!(grouped_history[gen], (worker=entry["worker"], points=entry["points"]))
end

#

xs = ys = range(-7, 7, length=500)
Z = [100 * (y - x^2)^2 + (1 - x)^2 for x in xs, y in ys]
rosenbrock(x, y) = 100 * (y - x^2)^2 + (1 - x)^2

#                    3D-                (
                          )
xs_3d = ys_3d = range(-7, 7, length=100)
Z_3d = [rosenbrock(x, y) for x in xs_3d, y in ys_3d]

#                    2D
anim_2d = @animate for gen in sort(collect(keys(grouped_history)))
    frame = grouped_history[gen]

    contourf(xs, ys, Z; c = :viridis, fillalpha = 0.3, legend = :topright,
             xlabel = "x", ylabel = "y", title = "Generation $gen (2D)")

    for (worker, points) in frame
        xs_points = [p[1] for p in points]
        ys_points = [p[2] for p in points]
        scatter!(xs_points, ys_points; color = worker_colors[worker], label = worker)
    end
end

gif(anim_2d, "evolution_2d.gif", fps = 10)
println("2D done")

#                    3D
anim_3d = @animate for gen in sort(collect(keys(grouped_history)))
    frame = grouped_history[gen]

    surface(xs_3d, ys_3d, Z_3d; xlabel="x", ylabel="y", zlabel="f(x, y)",
             title="Generation $gen (3D)", camera=(30, 60), legend = :topright, c = :viridis, fillalpha = 0.7)

    for (worker, points) in frame
        xs_points = [p[1] for p in points]
        ys_points = [p[2] for p in points]
```

```
58          zs_points = [rosenbrock(p[1], p[2]) for p in points] #
                            Z
59          scatter!(xs_points, ys_points, zs_points; color = worker_colors[
        worker], label = worker, markersize = 4)
60      end
61  end
62
63  #                           GIF
64  gif(anim_3d, "evolution_3d.gif", fps = 10)
65  println("3D done")
```

Листинг 5: run.sh

```
1   #!/bin/bash
2
3   ISO_TIME=20
4
5   N_WORKERS=4
6   PORT=8082
7
8   julia master.jl &
9
10  for (( i=1; i<=N_WORKERS; i++ ))
11  do
12    julia worker.jl &
13  done
14
15  wait
```

3 Результаты

Результаты запуска представлены на рисунках **??**.

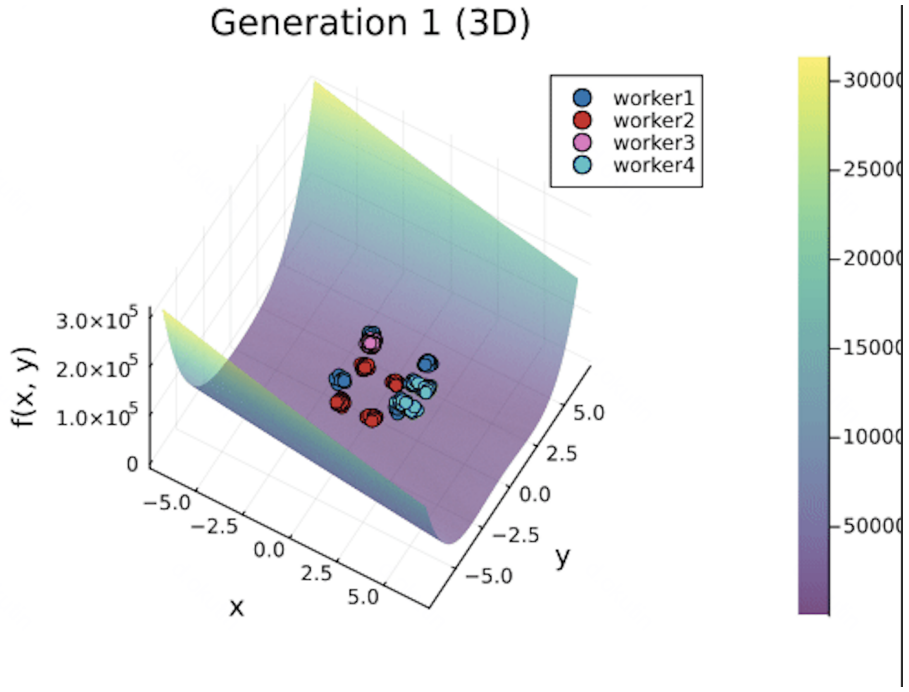

Рис. 1 — Визуализация (поколение 1)

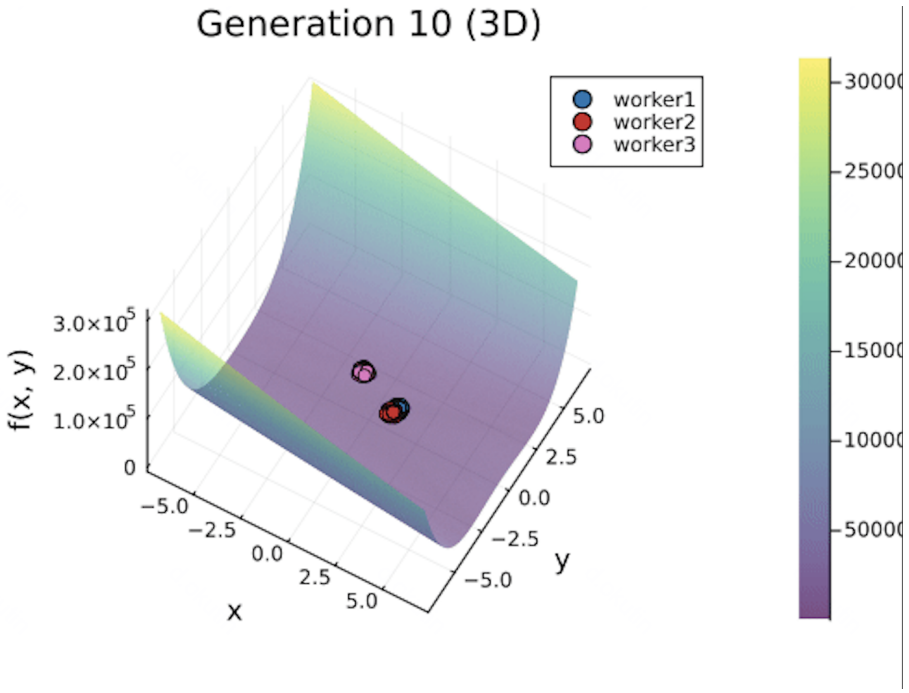

Рис. 2 — Визуализация (поколение 10)

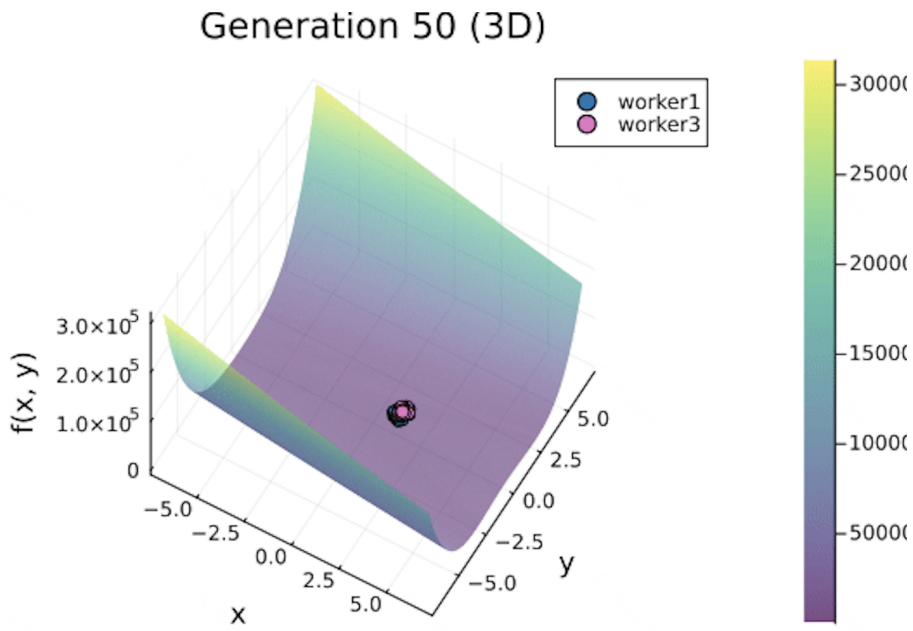

Рис. 3 — Визуализация (поколение 50)

4 Выводы

В ходе выполнения лабораторной работы было проведено сравнение классического и параллельного генетического алгоритмов с миграцией. Основные результаты и наблюдения:

- **Скорость работы**: Параллельный алгоритм показал большее время выполнения для данной задачи (оптимизация функции Розенброка с n = 2 n=2). Это обусловлено накладными расходами на передачу данных между узлами через WebSockets, включая сериализацию/десериализацию и сетевые задержки. Для простых задач эти издержки становятся критичными, снижая общую эффективность.

- **Качество решения**: Параллельный алгоритм продемонстрировал более разнообразное распределение особей благодаря миграциям, что снижает риск преждевременной сходимости к локальным минимумам. На визуализациях (Рис. 1–3) видно, как особи из разных рабочих узлов взаимодействуют, улучшая глобальный поиск.

- **Масштабируемость**: Для сложных задач (например, с высокой размерностью генов, большими популяциями или длительными вычислениями)

параллельный подход становится предпочтительным. Распределение вычислений между узлами позволяет сократить общее время работы, компенсируя накладные расходы. Кроме того, увеличение числа рабочих узлов (N_{workers}) улучшает масштабируемость алгоритма.

- **Рекомендации**:
  - Для задач с низкой вычислительной сложностью целесообразно использовать классический генетический алгоритм.
  - Для ресурсоемких задач, требующих параллелизации, миграционный алгоритм обеспечивает баланс между временем и качеством решения.
  - Оптимизация параметров (например, уменьшение ISOLATION_TIME или использование более эффективных протоколов передачи данных) может повысить производительность параллельной реализации.

Таким образом, несмотря на снижение скорости для малозатратных задач, параллельный генетический алгоритм с миграцией является мощным инструментом для решения сложных оптимизационных проблем, где важны как качество решения, так и возможность распределения вычислений.