

Министерство науки и высшего образования Российской Федерации  
Федеральное государственное автономное образовательное учреждение высшего образования «Уральский федеральный университет имени первого Президента России Б.Н. Ельцина» (УрФУ)

Физико-технологический институт

Кафедра «Технической физики»

Оценка

Преподаватель

Кашин И.В.

**ПАРАЛЛЕЛИЗАЦИЯ АЛГОРИТМА ДИНАМИКИ С ПРЕДПОЧТИТЕЛЬНЫМ РАСПРЕДЕЛЕНИЕМ ДЛЯ ВОСПРОИЗВЕДЕНИЯ ИЗОБРАЖЕНИЙ**

ОТЧЕТ

по лабораторной работе №6

|  |  |  |
| --- | --- | --- |
| Студент | Черняков Матвей Сергеевич | ФИО студента |

|  |
| --- |
| Специальность (направление подготовки) |
| 09.03.02 Информационные системы и технологии | |

|  |  |
| --- | --- |
| Группа | Фт-420008 |

Екатеринбург

2025

**ОГЛАВЛЕНИЕ**

[ОПИСАНИЕ ЗАДАЧИ 3](#_Toc211893056)

[ПРИНЦИП ПАРАЛЛЕЛИЗАЦИИ 7](#_Toc211893057)

[РЕШЕНИЕ ЗАДАЧИ 9](#_Toc211893058)

[ОЦЕНКА ЭФФЕКТИВНОСТИ ПАРАЛЛЕЛИЗАЦИИ 14](#_Toc211893059)

[ПРИЛОЖЕНИЕ А – ЛИСТИНГ КОДА 16](#_Toc211893060)

[ПРИЛОЖЕНИЕ Б – ЭКСПЕРИМЕНТ 1 32](#_Toc211893076)

[ПРИЛОЖЕНИЕ В – ЭКСПЕРИМЕНТ 2 33](#_Toc211893077)

[ПРИЛОЖЕНИЕ Г – РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ 35](#_Toc211893079)

ОПИСАНИЕ ЗАДАЧИ

Целью данной лабораторной работы является изучение одного из подходов к работе с изображениями с использованием **эвристических алгоритмов поиска**, основанных на динамике агентов в пространстве. Основная задача заключается в исследовании зависимости времени выполнения алгоритма от количества используемых процессов.

Эвристические (или метаэвристические) алгоритмы представляют собой класс методов оптимизации, которые используются для решения задач в пространствах с большим числом возможных вариантов, где полный перебор невозможен или непрактичен. Основная идея таких алгоритмов заключается в использовании локальной информации для принятия решений, которые в совокупности приводят к глобально оптимальному или близкому к оптимальному решению.

Алгоритм основан на фундаментальной идее: превращение независимых чисел во взаимосвязанные через **нормирование** (преобразование в проценты или вероятности).

Рассмотрим простой пример. Если у нас есть четыре независимых числа: 2, 3, 6, 8, они никак не связаны между собой. Изменение одного не влияет на другие. Но если мы превратим их в распределение (проценты от суммы):

Сумма = 2 + 3 + 6 + 8 = 19

Распределение: 2/19, 3/19, 6/19, 8/19

Теперь любое изменение любого числа автоматически влияет на все остальные через изменение знаменателя.

В алгоритме используется мультиагентный подход. Агенты – это абстрактные сущности, которые: имеют позицию в пространстве (для 2D изображения: координаты x, y). Могут перемещаться по дискретной сетке (8 направлений: вверх, вниз, влево, вправо и по диагоналям). Принимают решения на основе локальной информации. Оставляют след в виде увеличения значения динамического распределения в точке посещения.

Коллективное поведение агентов приводит к формированию целевого распределения. Каждый агент работает независимо, но через общее динамическое распределение все агенты косвенно влияют друг на друга.

Алгоритм оперирует двумя ключевыми структурами данных:

1. Эталонное распределение n̄(x,y)

Эталонное распределение – это целевое распределение, которое необходимо воспроизвести. В контексте задачи обработки изображений создается из исходного изображения путем преобразования яркости каждого пикселя в число. Изображение переводится в градации серого (grayscale). Каждый пиксель с координатами (x, y) имеет яркость в диапазоне [0, 255]. Значение яркости преобразуется: n̄(x,y) = яркость + 1.

Норма эталонного распределения:

Эталонное распределение показывает, сколько "визитов" агентов должно быть в каждой точке изображения. Темные области (малая яркость) требуют больше визитов, светлые (высокая яркость) — меньше.

2. Динамическое распределение (m(x,y))

Динамическое распределение – это изменяемая структура, которая накапливает информацию о перемещениях агентов. Инициализируется нулями (кроме начальных позиций агентов) m(x,y) = количество раз, которое агенты посетили точку (x, y). Обновляется на каждом шаге: когда агент перемещается в точку (x, y), значение m(x,y) увеличивается на 1.

Норма динамического распределения:

Динамическое распределение — это карта посещений агентов. Оно постепенно приближается к эталонному распределению в нормированном виде.

Для оценки качества воспроизведения используется **метрика различия** между эталонным и динамическим распределениями. В данной работе применяется нормированная относительная ошибка:

Метрика = 1000 × mean(|n̄(x,y) - (N̄/M) × m(x,y)|) / (n̄(x,y) + ε)

где: (N̄/M) × m(x,y)` – нормированное динамическое распределение

ε = 10⁻¹⁰ – малая константа для избежания деления на ноль

mean() – среднее по всем пикселям

Множитель 1000 используется для масштабирования в удобный диапазон.

Ключевой элемент алгоритма – правило перемещения агента. На каждом шаге агент:

1. Оценивает соседние позиции (до 8 направлений)

2. Вычисляет коэффициент K для каждой пробной позиции ():

3. Выбирает направление с максимальным значением K

4. Перемещается в выбранную позицию

5. Обновляет динамическое распределение:

K > 0 (недобор): В данной точке динамическое распределение меньше эталонного → агент стремится туда переместиться

K < 0 (перебор): В данной точке динамическое распределение больше эталонного → агент избегает этой точки

K = 0 (баланс): Точка находится в равновесии

Важная деталь реализации: 8 возможных направлений проверяются **в случайном порядке** на каждом шаге. Это достигается через:

1. Предварительную генерацию массива случайных перестановок   
   чисел 0-7
2. Использование новой перестановки на каждом шаге агента
3. Циклическое переиспользование массива перестановок

Несмотря на то, что каждый агент принимает решения на основе локальной информации (только значения в соседних клетках), коллективное поведение всех агентов приводит к **глобальному результату** — воспроизведению всего изображения.

Механизм достижения глобального эффекта:

1. Нормирование создает взаимосвязь: изменение M (общей нормы) влияет на все вычисления K во всех точках пространства
2. Отрицательная обратная связь: если агенты слишком долго находятся в одной области, K там становится отрицательным → агенты уходят
3. Самоорганизация: система естественным образом стремится к равномерному покрытию согласно эталонному распределению

Это позволяет алгоритму избегать локальных экстремумов — классической проблемы эвристических методов.

ПРИНЦИП ПАРАЛЛЕЛИЗАЦИИ

Ключевой задачей данной лабораторной работы является реализация и параллелизация алгоритма динамики с предпочтительным распределением для воспроизведения изображений с последующим анализом эффективности параллельной версии.

Для решения поставленной проблемы была выбрана гибридная стратегия, комбинирующая несколько подходов:

1. Использование **Shared Memory** (разделяемая память)

Реализация через multiprocessing.RawArray:

self.shared\_dynamic\_base = RawArray(ctypes.c\_double, int(self.height \* self.width))  
self.dynamic\_distribution = np.frombuffer(  
 self.shared\_dynamic\_base, dtype=np.float64  
).reshape((self.height, self.width))

Все процессы работают с одной областью памяти. Чтение данных происходит напрямую из RAM. Нет дублирования массивов для каждого процесса

2. **Пространственное разделение агентов**

Разделение агентов на группы, каждая из которых обрабатывается своим процессом.

Для 100 агентов и 4 процессов:

Процесс 1: агенты 0-24

Процесс 2: агенты 25-49

Процесс 3: агенты 50-74

Процесс 4: агенты 75-99

Реализация:

agents\_per\_process = self.n\_agents // self.n\_processes  
tasks = []  
  
for i in range(self.n\_processes):  
 agent\_start = i \* agents\_per\_process  
 if i == self.n\_processes - 1:  
 agent\_end = self.n\_agents  
 else:  
 agent\_end = (i + 1) \* agents\_per\_process

Каждый процесс имеет фиксированную группу агентов. Нет конфликтов при обновлении позиций агентов. Равномерная нагрузка между процессами.

3. Батчевая (**batch**) **обработка** с локальной агрегацией

Проблема race conditions:

При параллельной работе возможна ситуация:

Процесс 1: читает m[50,50] = 100

Процесс 2: читает m[50,50] = 100 ← то же значение

Процесс 1: пишет m[50,50] = 101

Процесс 2: пишет m[50,50] = 101 ← возможная потеря обновления

Решение – локальные обновления:

local\_dynamic\_updates = {} # словарь: (y,x) -> количество визитов

for step in range(n\_steps):

for agent\_idx in my\_agents:

# Вычисляем новую позицию

new\_x, new\_y = move\_agent(agent\_idx)

# Накапливаем изменения локально (БЕЗ записи в shared memory)

local\_dynamic\_updates[(new\_y, new\_x)] += 1

После завершения всех шагов возвращаем агрегированные изменения

return local\_dynamic\_updates

4. **Переиспользование Pool процессов**

Проблема overhead создания процессов:

# создание Pool на каждый вызов

def step(n\_steps):

with Pool(n\_processes) as pool: # Создание ~1-2 секунды!

pool.map(worker, tasks)

Решение:

class OptimizedParallelDPD:

def \_\_init\_\_(self, ...):

# Создаем Pool ОДИН раз при инициализации

self.pool = Pool(

processes=self.n\_processes,

initializer=init\_worker\_shared,

initargs=(shared\_memory\_args...)

)

def parallel\_step(self, n\_steps):

# Переиспользуем существующий Pool

results = self.pool.map(worker\_step\_batch, tasks)

def \_\_del\_\_(self):

# Корректно закрываем при удалении

self.pool.close()

self.pool.join()

РЕШЕНИЕ ЗАДАЧИ

Класс DynamicsWithPreferredDistribution – ключевой класс, реализующий функции чтения картинки, запуска агентов и поиска распределения близкого к исходному.

class DynamicsWithPreferredDistribution:  
 *"""  
 Алгоритм динамики с предпочтительным распределением  
 для воспроизведения изображений  
 """*

Функция init\_worker\_shared – функция-инициатор воркера с общей памятью для всех агентов.

def init\_worker\_shared(dynamic\_base, target\_base, orders\_base, M\_counter\_base, shape, n\_orders, directions\_tuple):  
 *"""Инициализация воркера с доступом к shared memory"""*

Функция worker\_step\_batch – воркер, обрабатывающий батч (часть) агентов.

def worker\_step\_batch(args):  
 *"""  
 Воркер обрабатывает батч агентов  
 """*

Класс OptimizedParallelDPD – класс, распаралелливающий алгоритм поиска агентами.

class OptimizedParallelDPD(DynamicsWithPreferredDistribution):  
 *"""  
 Оптимизированная параллельная версия  
 """*

Далее для экспериментов используется вызов этих функций и функций воркеров с помощью:

*# Запускаем для каждого числа процессов*for n\_proc in range(1, max\_processes + 1):  
 print(f"\n--- Запуск на {n\_proc} процессе(ах) ---")  
 exec\_time, algo = run\_single\_experiment(  
 image\_path,  
 n\_proc,  
 iterations,  
 checkpoint\_iterations=checkpoints,  
 verbose=True  
 )  
 results[n\_proc] = exec\_time

За исходное изображение было выбрано изображение 44-го президента США – Барака Обамы, которое было приведено к черно-белому фильтру и масштабировано к размеру 500х500 пикселей.



Рисунок 1 – Исходное изображение

При выполнении задачи были получены следующие результаты:

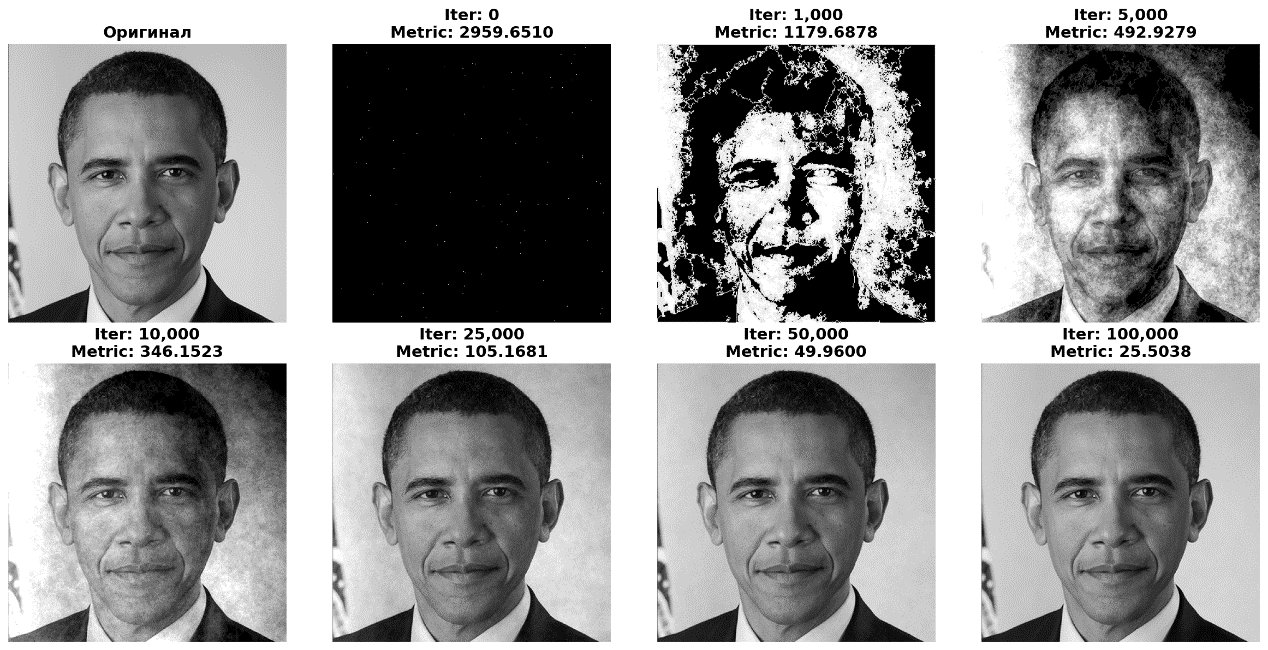


Рисунок 2 – Результаты при M=1, Iter=100,000

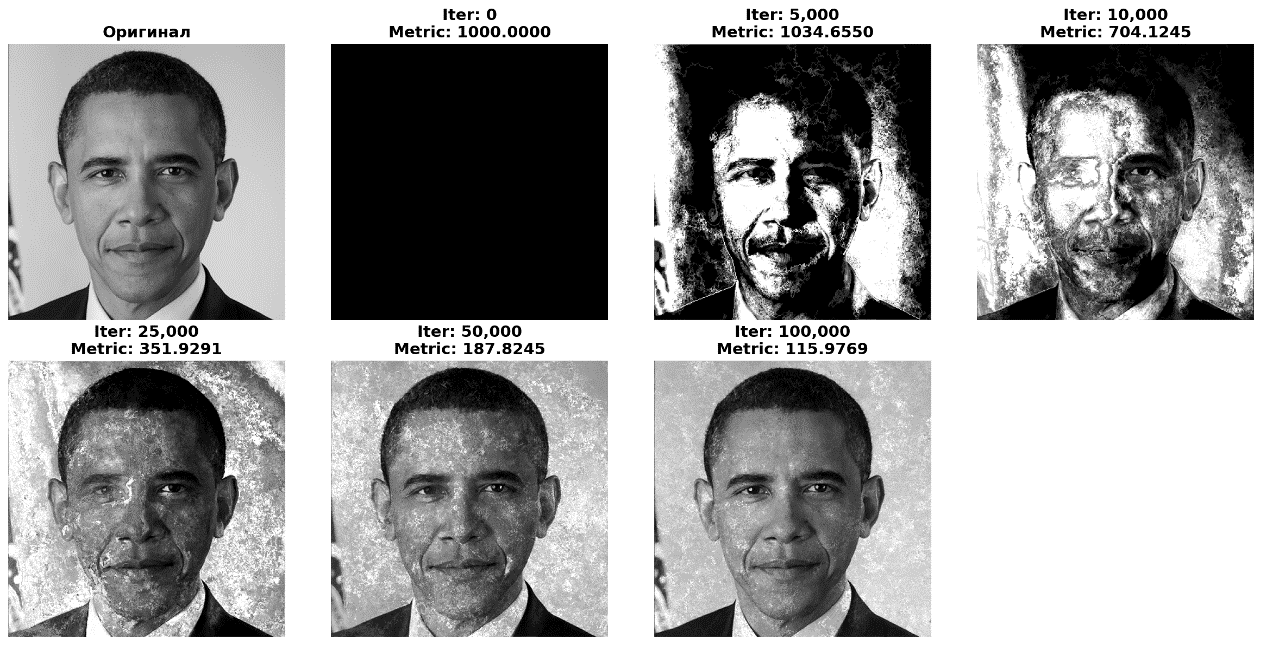


Рисунок 3 – Результаты при M=8, Iter=100,000

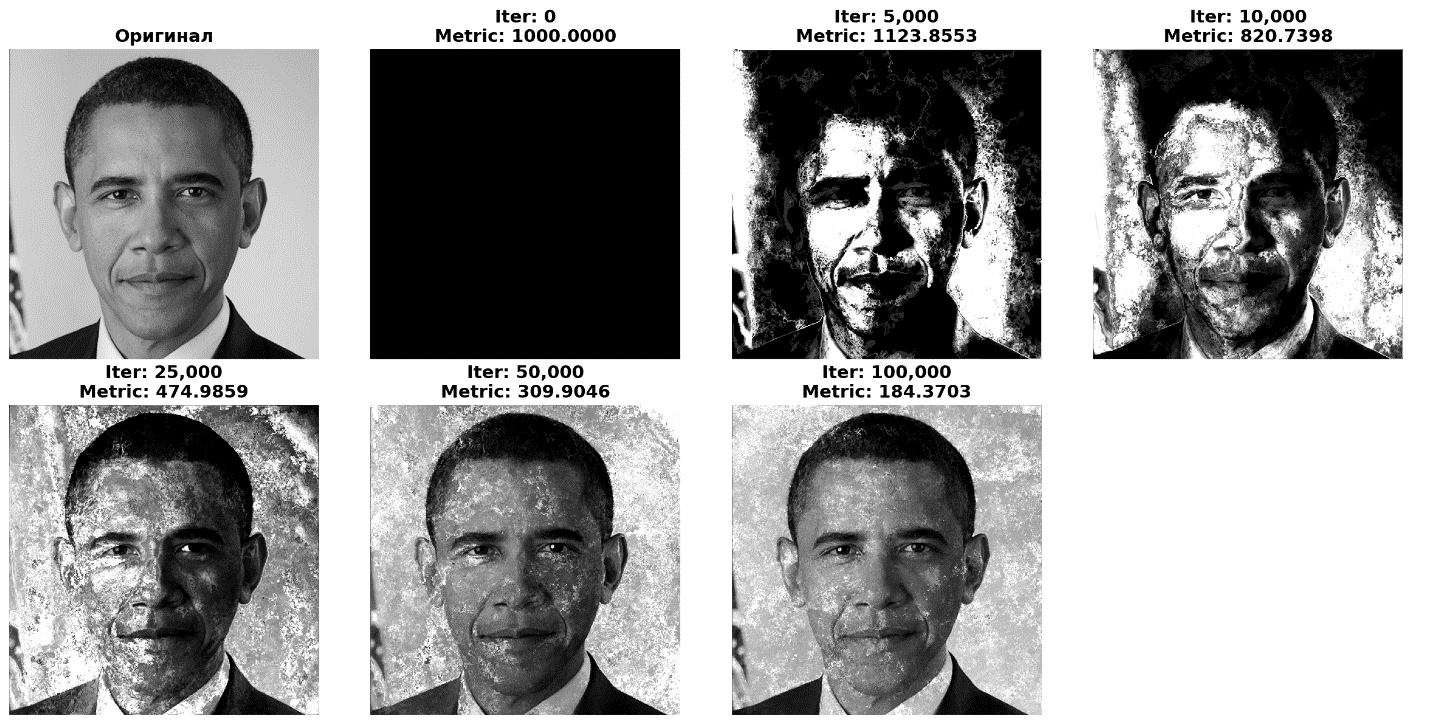


Рисунок 4 – Результаты при M=24, Iter=100,000

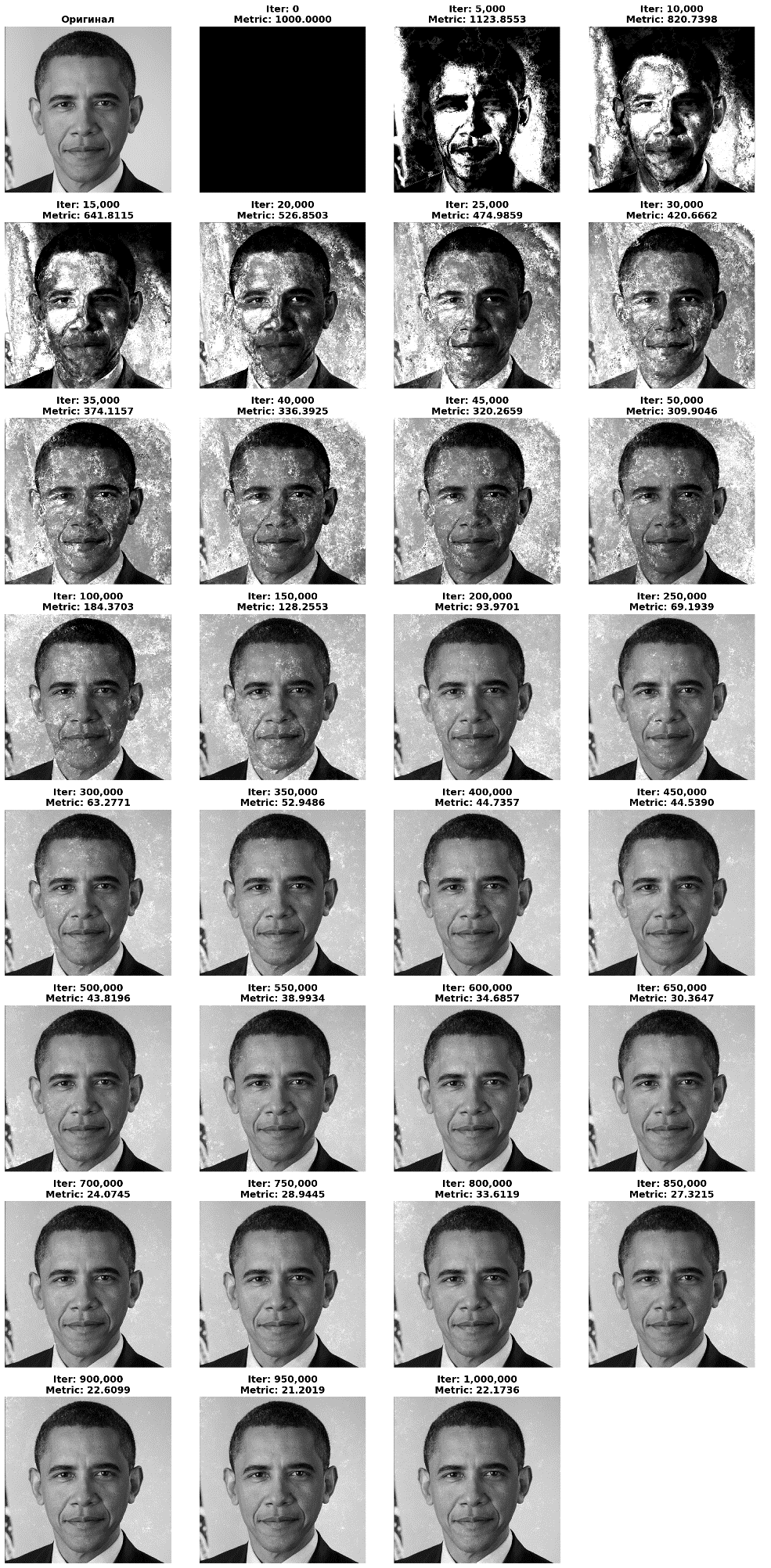


Рисунок 5 – Результаты при M=24, Iter=1,000,000

ОЦЕНКА ЭФФЕКТИВНОСТИ ПАРАЛЛЕЛИЗАЦИИ

Ключевыми параметрами является количество ядер (12) и количество потоков (24). Для данных характеристик было проведено тестирование сравнения идеального и условного времени при M от 1 до 24. Результаты тестирования представлены на графике (рисунок 6). Точные значения представлены в приложении Б;

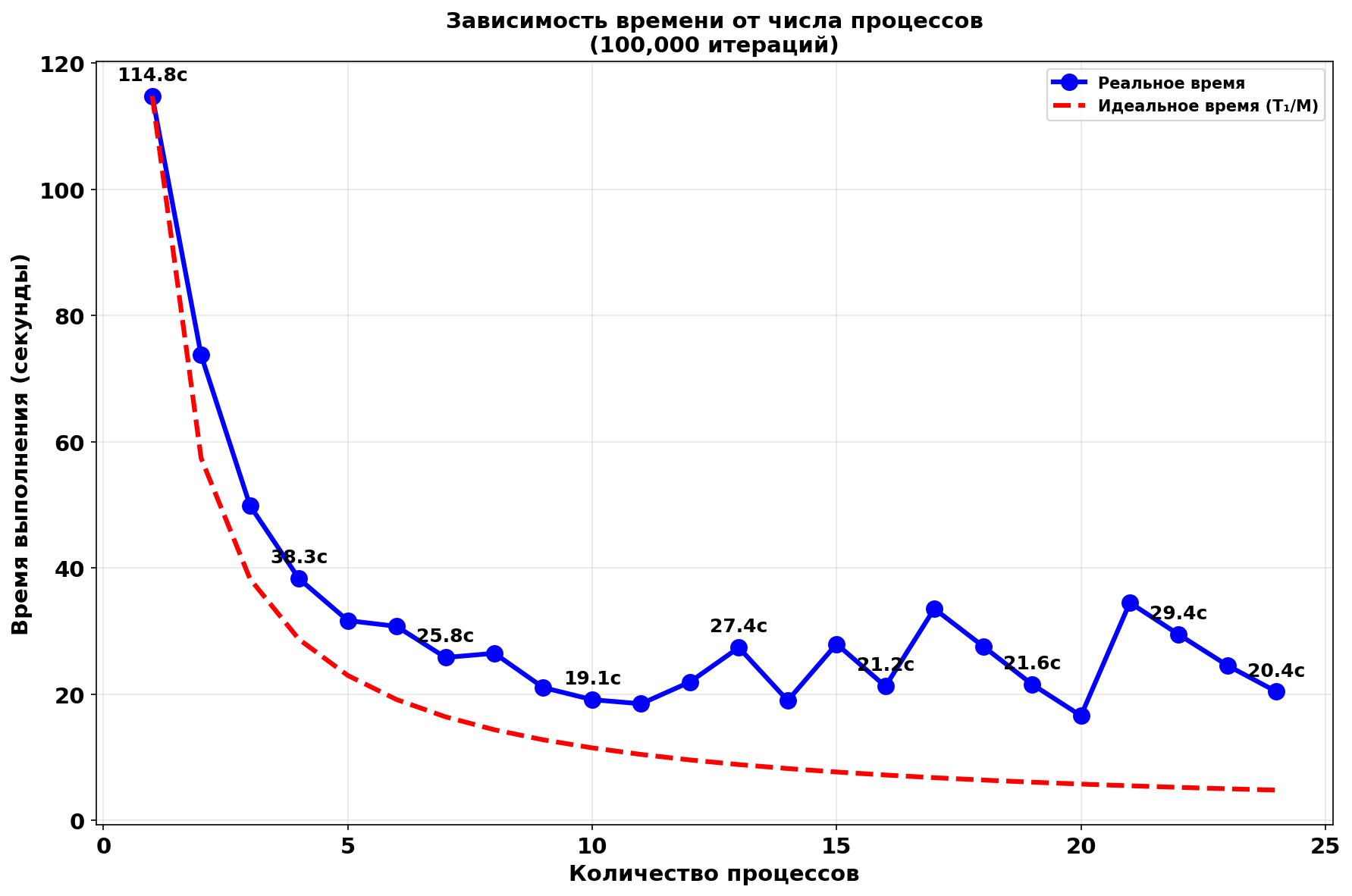


Рисунок 6 – График эксперимента 1

По графику для эксперимента 1 видно, что реальное время достаточно близко к идеальному. Отсюда можно понять, что можно выбрать оптимальное количество потоков для распараллеливания данного алгоритма сортировки (здесь это 20 потоков).

Для второго эксперимента проводилось постепенное увеличение количества итераций при 24 потоках. График на рисунке 7 показывает, что при увеличении количества итераций ошибка снижается и стремится к нулю, а это значит, что изображение становится ближе к оригинальному.

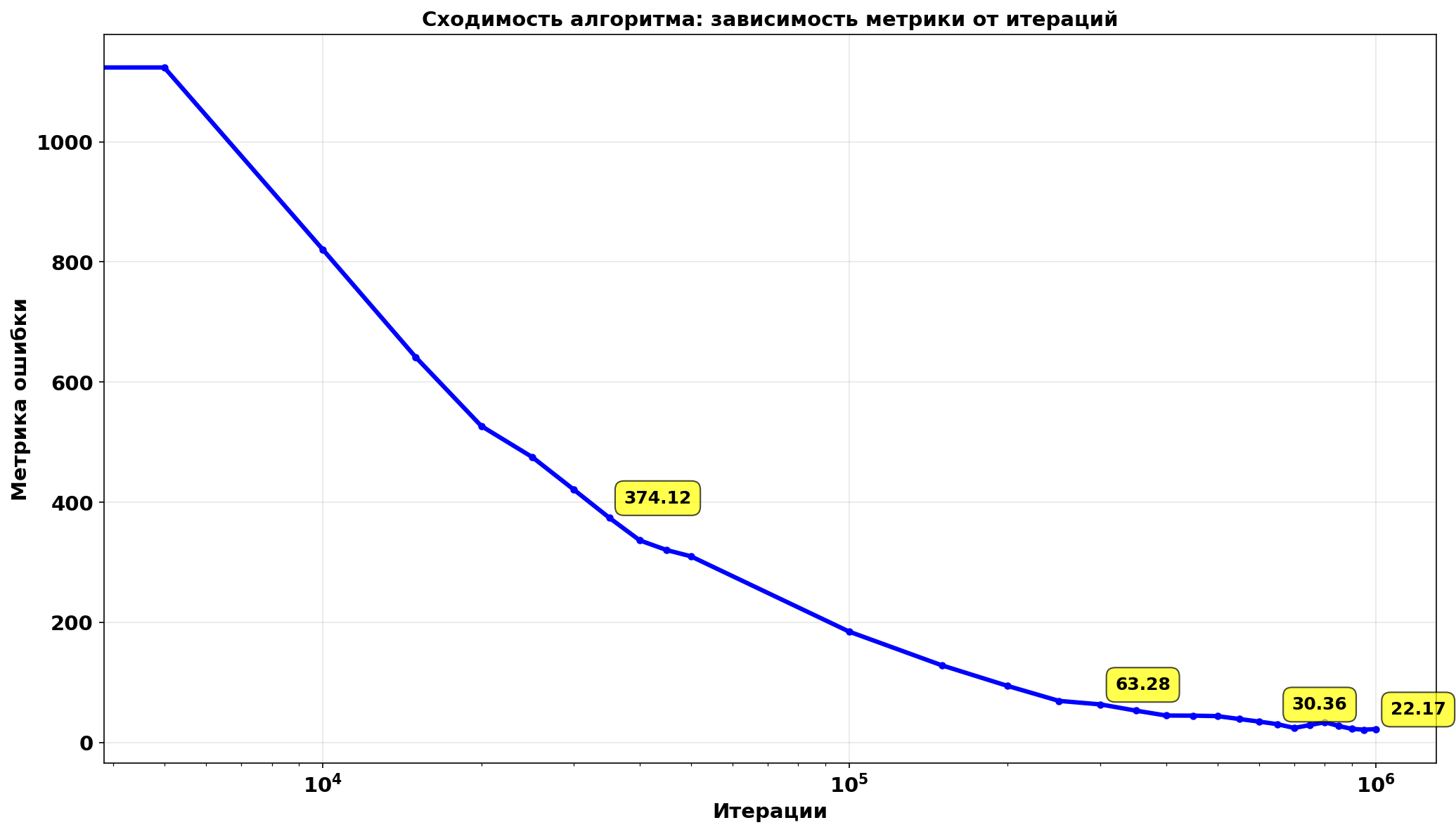


Рисунок 7 – График эксперимента 2

ПРИЛОЖЕНИЕ А – ЛИСТИНГ КОДА

import numpy as np  
from PIL import Image  
import matplotlib.pyplot as plt  
from multiprocessing import Pool, cpu\_count, RawArray  
import time  
from typing import Tuple, List  
import os  
import ctypes  
  
  
class DynamicsWithPreferredDistribution:  
 *"""  
 Алгоритм динамики с предпочтительным распределением  
 для воспроизведения изображений  
 """* def \_\_init\_\_(self, image\_path: str, n\_agents: int = 100, seed: int = 42):  
 *"""  
 Инициализация алгоритма  
  
 Args:  
 image\_path: путь к изображению  
 n\_agents: количество агентов  
 seed: seed для воспроизводимости  
 """* np.random.seed(seed)  
  
 *# Загрузка и подготовка эталонного распределения* self.load\_image(image\_path)  
  
 *# Параметры* self.n\_agents = n\_agents  
 self.height, self.width = self.target\_distribution.shape  
  
 *# Инициализация динамического распределения* self.dynamic\_distribution = np.zeros((self.height, self.width), dtype=np.float64)  
 self.M = 0 *# норма динамического распределения  
  
 # Инициализация агентов в случайных позициях* self.agents\_x = np.random.randint(0, self.width, n\_agents)  
 self.agents\_y = np.random.randint(0, self.height, n\_agents)  
  
 *# Размещаем агентов на динамическом распределении* for x, y in zip(self.agents\_x, self.agents\_y):  
 self.dynamic\_distribution[y, x] += 1  
 self.M += 1  
  
 *# Подготовка случайных порядков проверки направлений* self.prepare\_random\_orders(10000)  
 self.current\_order\_idx = 0

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

*# 8 направлений: (dx, dy)* self.directions = [  
 (-1, -1), (0, -1), (1, -1),  
 (-1, 0), (1, 0),  
 (-1, 1), (0, 1), (1, 1)  
 ]  
  
 *# История для визуализации* self.history = []  
  
 *# История метрик для графиков* self.metric\_history = []  
  
 def load\_image(self, image\_path: str):  
 *"""Загрузка изображения и создание эталонного распределения"""* img = Image.open(image\_path).convert('L') *# в grayscale  
  
 # Изменяем размер если нужно* max\_size = 400  
 if max(img.size) > max\_size:  
 ratio = max\_size / max(img.size)  
 new\_size = (int(img.size[0] \* ratio), int(img.size[1] \* ratio))  
 *# Совместимость с разными версиями Pillow* try:  
 img = img.resize(new\_size, Image.Resampling.LANCZOS)  
 except AttributeError:  
 img = img.resize(new\_size, Image.LANCZOS)  
  
 *# Преобразуем в numpy массив* img\_array = np.array(img, dtype=np.float64)  
  
 *# Нормализуем яркость к диапазону [1, 256]* self.target\_distribution = img\_array + 1.0  
  
 *# Вычисляем норму эталонного распределения* self.N\_target = np.sum(self.target\_distribution)  
  
 print(f"Изображение загружено: {self.target\_distribution.shape}")  
 print(f"Норма эталонного распределения: {self.N\_target:.2f}")  
  
 def prepare\_random\_orders(self, n\_orders: int = 10000):  
 *"""Подготовка случайных порядков проверки направлений"""* self.random\_orders = np.zeros((n\_orders, 8), dtype=np.int32)  
 for i in range(n\_orders):  
 self.random\_orders[i] = np.random.permutation(8)  
  
 def get\_next\_order(self) -> np.ndarray:  
 *"""Получить следующий случайный порядок проверки"""* order = self.random\_orders[self.current\_order\_idx]  
 self.current\_order\_idx = (self.current\_order\_idx + 1) % len(self.random\_orders)  
 return order

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

def calculate\_k(self, x: int, y: int) -> float:  
 *"""  
 Вычисление коэффициента K для точки (x, y)  
 K = n\_target(x,y) - (N\_target/M) \* m(x,y)  
 """* if self.M == 0:  
 return self.target\_distribution[y, x]  
  
 normalized\_dynamic = (self.N\_target / self.M) \* self.dynamic\_distribution[y, x]  
 return self.target\_distribution[y, x] - normalized\_dynamic  
  
 def move\_agent(self, agent\_idx: int) -> Tuple[int, int]:  
 *"""  
 Перемещение одного агента  
  
 Returns:  
 новые координаты (x, y)  
 """* x, y = self.agents\_x[agent\_idx], self.agents\_y[agent\_idx]  
  
 *# Получаем случайный порядок проверки направлений* order = self.get\_next\_order()  
  
 best\_k = float('-inf')  
 best\_x, best\_y = x, y  
  
 *# Проверяем все направления в случайном порядке* for dir\_idx in order:  
 dx, dy = self.directions[dir\_idx]  
 new\_x, new\_y = x + dx, y + dy  
  
 *# Проверка границ* if 0 <= new\_x < self.width and 0 <= new\_y < self.height:  
 k = self.calculate\_k(new\_x, new\_y)  
 if k > best\_k:  
 best\_k = k  
 best\_x, best\_y = new\_x, new\_y  
  
 return best\_x, best\_y  
  
 def step(self, n\_steps: int = 1):  
 *"""Выполнить n\_steps шагов алгоритма"""* for \_ in range(n\_steps):  
 *# Перемещаем всех агентов последовательно* for agent\_idx in range(self.n\_agents):  
 new\_x, new\_y = self.move\_agent(agent\_idx)  
  
 *# Обновляем позицию агента* self.agents\_x[agent\_idx] = new\_x  
 self.agents\_y[agent\_idx] = new\_y  
  
 *# Обновляем динамическое распределение* self.dynamic\_distribution[new\_y, new\_x] += 1  
 self.M += 1

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

def calculate\_metric(self) -> float:  
 *"""  
 Вычисление метрики различия между распределениями  
 Используем нормированную среднеквадратичную ошибку  
 """* if self.M == 0:  
 return float('inf')  
  
 *# Нормируем динамическое распределение* normalized\_dynamic = (self.N\_target / self.M) \* self.dynamic\_distribution  
  
 *# Вычисляем относительную ошибку* diff = np.abs(self.target\_distribution - normalized\_dynamic)  
 relative\_error = np.mean(diff / (self.target\_distribution + 1e-10))  
  
 return relative\_error \* 1000 *# Масштабируем для читаемости* def get\_current\_image(self) -> np.ndarray:  
 *"""Получить текущее изображение из динамического распределения"""* if self.M == 0:  
 return np.zeros\_like(self.target\_distribution)  
  
 *# Нормируем динамическое распределение* normalized = (self.N\_target / self.M) \* self.dynamic\_distribution  
  
 *# Возвращаем как есть (без инверсии)* result = np.clip(normalized - 1.0, 0, 255)  
  
 return result  
  
 def save\_checkpoint(self, iteration: int, save\_to\_history: bool = True):  
 *"""Сохранить контрольную точку"""* current\_img = self.get\_current\_image()  
 metric = self.calculate\_metric()  
  
 if save\_to\_history:  
 self.history.append({  
 'iteration': iteration,  
 'image': current\_img.copy(),  
 'metric': metric,  
 'M': self.M  
 })  
  
 *# Всегда сохраняем в metric\_history для графиков* self.metric\_history.append({  
 'iteration': iteration,  
 'metric': metric  
 })

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

print(f"Checkpoint - Итерация {iteration:,}: метрика = {metric:.4f}, M = {self.M:,}")  
  
 def visualize\_progress(self, save\_path: str = 'progress.png'):  
 *"""Визуализация прогресса воспроизведения"""* n\_checkpoints = len(self.history)  
 if n\_checkpoints == 0:  
 print("Нет данных для визуализации")  
 return  
  
 *# Создаем сетку для отображения (добавляем +1 для оригинала)* n\_cols = min(4, n\_checkpoints + 1)  
 n\_rows = (n\_checkpoints + 1 + n\_cols - 1) // n\_cols *# Округление вверх* fig, axes = plt.subplots(n\_rows, n\_cols, figsize=(n\_cols \* 4, n\_rows \* 4))  
 if n\_rows == 1 and n\_cols == 1:  
 axes = np.array([axes])  
 elif n\_rows == 1:  
 axes = axes.reshape(1, -1)  
 elif n\_cols == 1:  
 axes = axes.reshape(-1, 1)  
 axes = axes.flatten()  
  
 *# Показываем оригинал* original\_img = self.target\_distribution - 1.0  
 axes[0].imshow(original\_img, cmap='gray', vmin=0, vmax=255)  
 axes[0].set\_title('Оригинал', fontsize=14, fontweight='bold')  
 axes[0].axis('off')  
  
 *# Показываем промежуточные результаты* for idx, checkpoint in enumerate(self.history):  
 axes[idx + 1].imshow(checkpoint['image'], cmap='gray', vmin=0, vmax=255)  
 axes[idx + 1].set\_title(f"Iter: {checkpoint['iteration']:,}\nMetric: {checkpoint['metric']:.4f}",  
 fontsize=14, fontweight='bold')  
 axes[idx + 1].axis('off')  
  
 *# Скрываем неиспользуемые оси* for idx in range(n\_checkpoints + 1, len(axes)):  
 axes[idx].axis('off')  
  
 plt.tight\_layout()  
 plt.savefig(save\_path, dpi=150, bbox\_inches='tight')  
 print(f"Визуализация сохранена в {save\_path}")  
 plt.close()  
  
  
*# Глобальные переменные для shared memory и Pool*shared\_dynamic = None  
shared\_target = None  
shared\_random\_orders = None  
shared\_directions = None  
shared\_M\_counter = None  
global\_pool = None

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

def init\_worker\_shared(dynamic\_base, target\_base, orders\_base, M\_counter\_base, shape, n\_orders, directions\_tuple):  
 *"""Инициализация воркера с доступом к shared memory"""* global shared\_dynamic, shared\_target, shared\_random\_orders, shared\_directions, shared\_M\_counter  
  
 *# Создаем numpy массивы из shared memory* shared\_dynamic = np.frombuffer(dynamic\_base, dtype=np.float64).reshape(shape)  
 shared\_target = np.frombuffer(target\_base, dtype=np.float64).reshape(shape)  
 shared\_random\_orders = np.frombuffer(orders\_base, dtype=np.int32).reshape((n\_orders, 8))  
 shared\_M\_counter = np.frombuffer(M\_counter\_base, dtype=np.int64)  
 shared\_directions = directions\_tuple  
  
  
def worker\_step\_batch(args):  
 *"""  
 Воркер обрабатывает батч агентов  
 """* agent\_start, agent\_end, agents\_x, agents\_y, n\_steps, order\_start\_idx, N\_target, height, width = args  
  
 *# Локальные копии позиций агентов* local\_agents\_x = agents\_x[agent\_start:agent\_end].copy()  
 local\_agents\_y = agents\_y[agent\_start:agent\_end].copy()  
  
 *# Локальное динамическое распределение для накопления изменений* local\_dynamic\_updates = {}  
  
 *# Читаем текущее M* current\_M = int(shared\_M\_counter[0])  
  
 order\_idx = order\_start\_idx  
  
 *# Обрабатываем n\_steps для наших агентов* for step in range(n\_steps):  
 for local\_idx in range(len(local\_agents\_x)):  
 x = local\_agents\_x[local\_idx]  
 y = local\_agents\_y[local\_idx]  
  
 *# Получаем порядок проверки* order = shared\_random\_orders[order\_idx % len(shared\_random\_orders)]  
 order\_idx += 1  
  
 best\_k = float('-inf')  
 best\_x, best\_y = x, y  
  
 *# Проверяем направления* for dir\_idx in order:  
 dx, dy = shared\_directions[dir\_idx]  
 new\_x, new\_y = x + dx, y + dy

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

if 0 <= new\_x < width and 0 <= new\_y < height:  
 *# Вычисляем K* if current\_M == 0:  
 k = shared\_target[new\_y, new\_x]  
 else:  
 dynamic\_val = shared\_dynamic[new\_y, new\_x]  
 if (new\_y, new\_x) in local\_dynamic\_updates:  
 dynamic\_val += local\_dynamic\_updates[(new\_y, new\_x)]  
  
 normalized\_dynamic = (N\_target / current\_M) \* dynamic\_val  
 k = shared\_target[new\_y, new\_x] - normalized\_dynamic  
  
 if k > best\_k:  
 best\_k = k  
 best\_x, best\_y = new\_x, new\_y  
  
 *# Обновляем локальную позицию* local\_agents\_x[local\_idx] = best\_x  
 local\_agents\_y[local\_idx] = best\_y  
  
 *# Накапливаем изменения локально* key = (best\_y, best\_x)  
 local\_dynamic\_updates[key] = local\_dynamic\_updates.get(key, 0) + 1  
 current\_M += 1  
  
 return agent\_start, agent\_end, local\_agents\_x, local\_agents\_y, local\_dynamic\_updates  
  
  
class OptimizedParallelDPD(DynamicsWithPreferredDistribution):  
 *"""  
 Оптимизированная параллельная версия  
 """* def \_\_init\_\_(self, image\_path: str, n\_agents: int = 100, n\_processes: int = None, seed: int = 42):  
 super().\_\_init\_\_(image\_path, n\_agents, seed)  
 self.n\_processes = n\_processes or cpu\_count()  
 print(f"Используется {self.n\_processes} процессов (оптимизированная версия)")  
  
 *# Создаем shared memory* self.\_init\_shared\_memory()  
  
 *# Создаем Pool один раз* self.pool = None  
 self.\_init\_pool()

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

def \_init\_shared\_memory(self):  
 *"""Инициализация shared memory"""  
 # Динамическое распределение* self.shared\_dynamic\_base = RawArray(ctypes.c\_double, int(self.height \* self.width))  
 self.dynamic\_distribution = np.frombuffer(  
 self.shared\_dynamic\_base, dtype=np.float64  
 ).reshape((self.height, self.width))  
  
 *# Целевое распределение* self.shared\_target\_base = RawArray(ctypes.c\_double, self.target\_distribution.flatten().tolist())  
  
 *# Случайные порядки* self.shared\_orders\_base = RawArray(ctypes.c\_int32, self.random\_orders.flatten().tolist())  
  
 *# Счетчик M* self.shared\_M\_base = RawArray(ctypes.c\_int64, 1)  
 self.shared\_M\_base[0] = self.M  
  
 def \_init\_pool(self):  
 *"""Инициализация Pool (создается один раз)"""* self.pool = Pool(  
 processes=self.n\_processes,  
 initializer=init\_worker\_shared,  
 initargs=(  
 self.shared\_dynamic\_base,  
 self.shared\_target\_base,  
 self.shared\_orders\_base,  
 self.shared\_M\_base,  
 (self.height, self.width),  
 len(self.random\_orders),  
 self.directions  
 )  
 )  
  
 def parallel\_step(self, n\_steps: int = 100):  
 *"""  
 Параллельное выполнение n\_steps шагов  
 """  
 # Обновляем M в shared memory* self.shared\_M\_base[0] = self.M  
  
 *# Разделяем агентов между процессами* agents\_per\_process = self.n\_agents // self.n\_processes  
 tasks = []  
  
 for i in range(self.n\_processes):  
 agent\_start = i \* agents\_per\_process  
 if i == self.n\_processes - 1:  
 agent\_end = self.n\_agents  
 else:  
 agent\_end = (i + 1) \* agents\_per\_process  
  
 order\_start = (self.current\_order\_idx + i \* 1000) % len(self.random\_orders)

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

tasks.append((  
 agent\_start,  
 agent\_end,  
 self.agents\_x,  
 self.agents\_y,  
 n\_steps,  
 order\_start,  
 self.N\_target,  
 self.height,  
 self.width  
 ))  
  
 *# Выполняем параллельно с переиспользованием Pool* results = self.pool.map(worker\_step\_batch, tasks)  
  
 *# Применяем обновления из всех процессов* for agent\_start, agent\_end, new\_agents\_x, new\_agents\_y, dynamic\_updates in results:  
 *# Обновляем позиции агентов* self.agents\_x[agent\_start:agent\_end] = new\_agents\_x  
 self.agents\_y[agent\_start:agent\_end] = new\_agents\_y  
  
 *# Применяем обновления динамического распределения* for (y, x), count in dynamic\_updates.items():  
 self.dynamic\_distribution[y, x] += count  
 self.M += count  
  
 *# Обновляем индекс порядка* self.current\_order\_idx = (self.current\_order\_idx + n\_steps \* self.n\_agents) % len(self.random\_orders)  
  
 def \_\_del\_\_(self):  
 *"""Закрываем Pool при удалении объекта"""* if hasattr(self, 'pool') and self.pool is not None:  
 self.pool.close()  
 self.pool.join()  
  
  
def run\_single\_experiment(image\_path: str, n\_processes: int, max\_iterations: int,  
 checkpoint\_iterations: list = None, verbose: bool = True):  
 *"""  
 Запуск одного эксперимента  
  
 Args:  
 checkpoint\_iterations: список итераций для чекпоинтов  
  
 Returns:  
 tuple: (execution\_time, algo\_instance)  
 """* if verbose:  
 print(f"\n{'=' \* 60}")  
 print(f"Эксперимент: {n\_processes} процесс(ов), {max\_iterations:,} итераций")  
 print(f"{'=' \* 60}")

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

*# Создаем алгоритм* if n\_processes == 1:  
 algo = DynamicsWithPreferredDistribution(image\_path, n\_agents=100)  
 step\_func = algo.step  
 batch\_size = 1000  
 else:  
 algo = OptimizedParallelDPD(image\_path, n\_agents=100, n\_processes=n\_processes)  
 step\_func = algo.parallel\_step  
 batch\_size = 5000  
  
 *# Начальный чекпоинт* algo.save\_checkpoint(0, save\_to\_history=True)  
  
 *# Определяем чекпоинты* if checkpoint\_iterations is None:  
 checkpoint\_iterations = [max\_iterations]  
  
 checkpoint\_set = set(checkpoint\_iterations)  
  
 start\_time = time.time()  
 current\_iteration = 0  
 last\_print\_time = start\_time  
  
 *# Основной цикл* while current\_iteration < max\_iterations:  
 *# Определяем размер шага* remaining = max\_iterations - current\_iteration  
 step\_size = min(batch\_size, remaining)  
  
 step\_func(step\_size)  
 current\_iteration += step\_size  
  
 *# Сохраняем чекпоинт если достигли нужной итерации* if current\_iteration in checkpoint\_set:  
 algo.save\_checkpoint(current\_iteration, save\_to\_history=True)  
  
 *# Вывод прогресса* if verbose and time.time() - last\_print\_time >= 3.0:  
 elapsed = time.time() - start\_time  
 speed = current\_iteration / elapsed if elapsed > 0 else 0  
 eta = (max\_iterations - current\_iteration) / speed if speed > 0 else 0  
 print(f" Прогресс: {current\_iteration:,}/{max\_iterations:,} | "  
 f"Скорость: {speed:.0f} шаг/с | ETA: {eta:.0f}с")  
 last\_print\_time = time.time()  
  
 total\_time = time.time() - start\_time  
  
 if verbose:  
 final\_metric = algo.calculate\_metric()  
 print(f"\nЗавершено за {total\_time:.2f}с")  
 print(f"Финальная метрика: {final\_metric:.4f}")  
 print(f"Средняя скорость: {max\_iterations / total\_time:.0f} шаг/с")

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

*# Закрываем Pool если это параллельная версия* if n\_processes > 1:  
 algo.pool.close()  
 algo.pool.join()  
  
 return total\_time, algo  
  
  
def experiment\_1\_scalability(image\_path: str, max\_processes: int = 24):  
 *"""  
 Эксперимент 1: Масштабируемость  
 Чекпоинты: 0, 1000, 2500, 5000, 10000, 25000, 50000, 100000  
 """* print("\n" + "=" \* 70)  
 print("ЭКСПЕРИМЕНТ 1: МАСШТАБИРУЕМОСТЬ (100,000 итераций)")  
 print("=" \* 70)  
  
 iterations = 100000  
 checkpoints = [0, 1000, 2500, 5000, 10000, 25000, 50000, 100000]  
 results = {}  
  
 *# Запускаем для каждого числа процессов* for n\_proc in range(1, max\_processes + 1):  
 print(f"\n--- Запуск на {n\_proc} процессе(ах) ---")  
 exec\_time, algo = run\_single\_experiment(  
 image\_path,  
 n\_proc,  
 iterations,  
 checkpoint\_iterations=checkpoints,  
 verbose=True  
 )  
 results[n\_proc] = exec\_time  
  
 *# Сохраняем визуализацию только для некоторых конфигураций* if n\_proc in [1, 4, 8, 16, max\_processes]:  
 algo.visualize\_progress(f'exp1\_progress\_{n\_proc}proc.png')  
  
 *# Строим график* plot\_scalability(results, iterations)  
  
 return results  
  
  
def experiment\_2\_convergence(image\_path: str, n\_processes: int = 24):  
 *"""  
 Эксперимент 2: Сходимость  
 Чекпоинты: каждые 500 до 50000, затем каждые 50000 до 1000000  
 """* print("\n" + "=" \* 70)  
 print(f"ЭКСПЕРИМЕНТ 2: СХОДИМОСТЬ (1,000,000 итераций на {n\_processes} процессах)")  
 print("=" \* 70)  
  
 max\_iterations = 1000000

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

*# Чекпоинты: каждые 500 до 50000, затем каждые 50000* checkpoints = list(range(0, 50001, 500)) + list(range(100000, 1000001, 50000))  
 checkpoints = sorted(set(checkpoints)) *# Убираем дубликаты и сортируем* exec\_time, algo = run\_single\_experiment(  
 image\_path,  
 n\_processes,  
 max\_iterations,  
 checkpoint\_iterations=checkpoints,  
 verbose=True  
 )  
  
 *# Строим график сходимости* plot\_convergence(algo.metric\_history)  
  
 *# Сохраняем визуализацию* algo.visualize\_progress('exp2\_progress\_convergence.png')  
  
 return exec\_time, algo  
  
  
def plot\_scalability(results: dict, iterations: int):  
 *"""  
 График масштабируемости (только время)  
 """* n\_processes = sorted(results.keys())  
 times = [results[n] for n in n\_processes]  
  
 *# Идеальное ускорение* t1 = times[0]  
 ideal\_times = [t1 / n for n in n\_processes]  
  
 *# Создаем фигуру с одним графиком* fig, ax = plt.subplots(1, 1, figsize=(12, 8))  
  
 *# График: Время выполнения* ax.plot(n\_processes, times, 'bo-', linewidth=3, markersize=10, label='Реальное время')  
 ax.plot(n\_processes, ideal\_times, 'r--', linewidth=3, label='Идеальное время (T₁/M)')  
 ax.set\_xlabel('Количество процессов', fontsize=14, fontweight='bold')  
 ax.set\_ylabel('Время выполнения (секунды)', fontsize=14, fontweight='bold')  
 ax.set\_title(f'Зависимость времени от числа процессов\n({iterations:,} итераций)',  
 fontsize=14, fontweight='bold')  
 ax.grid(True, alpha=0.3)  
 ax.legend(fontsize=14, prop={'weight': 'bold'})  
  
 *# Делаем метки осей жирными* ax.tick\_params(axis='both', labelsize=14)  
 for label in ax.get\_xticklabels() + ax.get\_yticklabels():  
 label.set\_fontweight('bold')

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

*# Добавляем значения на график* for i, (n, t) in enumerate(zip(n\_processes, times)):  
 if i % 3 == 0 or n == n\_processes[-1]:  
 ax.annotate(f'{t:.1f}с',  
 xy=(n, t),  
 xytext=(0, 10),  
 textcoords='offset points',  
 ha='center',  
 fontsize=12,  
 fontweight='bold')  
  
 plt.tight\_layout()  
 plt.savefig('experiment1\_scalability.png', dpi=150, bbox\_inches='tight')  
 print(f"\nГрафик масштабируемости сохранен: experiment1\_scalability.png")  
 plt.close()  
  
 *# Выводим таблицу результатов* speedups = [t1 / t for t in times]  
 print("\n" + "=" \* 70)  
 print("РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТА 1")  
 print("=" \* 70)  
 print(f"{'Процессы':<12} {'Время (с)':<12} {'Ускорение':<12} {'Эффективность':<15}")  
 print("-" \* 70)  
 for n, t, s in zip(n\_processes, times, speedups):  
 efficiency = (s / n) \* 100  
 print(f"{n:<12} {t:>10.2f} {s:>10.2f}x {efficiency:>12.1f}%")  
 print("=" \* 70)  
  
  
def plot\_convergence(metric\_history: list):  
 *"""  
 График сходимости  
 """* iterations = [item['iteration'] for item in metric\_history]  
 metrics = [item['metric'] for item in metric\_history]  
  
 *# Создаем график* fig, ax = plt.subplots(figsize=(14, 8))  
  
 ax.plot(iterations, metrics, 'b-', linewidth=3, marker='o', markersize=4)  
 ax.set\_xlabel('Итерации', fontsize=14, fontweight='bold')  
 ax.set\_ylabel('Метрика ошибки', fontsize=14, fontweight='bold')  
 ax.set\_title('Сходимость алгоритма: зависимость метрики от итераций',  
 fontsize=14, fontweight='bold')  
 ax.grid(True, alpha=0.3)  
  
 *# Логарифмическая шкала для итераций* ax.set\_xscale('log')  
  
 *# Делаем метки осей жирными* ax.tick\_params(axis='both', labelsize=14)  
 for label in ax.get\_xticklabels() + ax.get\_yticklabels():  
 label.set\_fontweight('bold')

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

*# Добавляем значения на ключевых точках* key\_indices = [0, len(iterations) // 4, len(iterations) // 2, 3 \* len(iterations) // 4, len(iterations) - 1]  
 for idx in key\_indices:  
 if idx < len(iterations):  
 ax.annotate(f'{metrics[idx]:.2f}',  
 xy=(iterations[idx], metrics[idx]),  
 xytext=(10, 10),  
 textcoords='offset points',  
 fontsize=12,  
 fontweight='bold',  
 bbox=dict(boxstyle='round,pad=0.5', facecolor='yellow', alpha=0.7))  
  
 plt.tight\_layout()  
 plt.savefig('experiment2\_convergence.png', dpi=150, bbox\_inches='tight')  
 print(f"\nГрафик сходимости сохранен: experiment2\_convergence.png")  
 plt.close()  
  
 *# Выводим  
 # Выводим таблицу результатов* print("\n" + "=" \* 70)  
 print("РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТА 2")  
 print("=" \* 70)  
 print(f"{'Итерация':<15} {'Метрика':<12} {'Улучшение от начала':<20}")  
 print("-" \* 70)  
  
 initial\_metric = metrics[0]  
 for i, m in zip(iterations, metrics):  
 improvement = ((initial\_metric - m) / initial\_metric) \* 100  
 print(f"{i:<15,} {m:>10.4f} {improvement:>18.1f}%")  
 print("=" \* 70)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 image\_path = "obama2.jpg"  
  
 *# Проверяем наличие изображения* if not os.path.exists(image\_path):  
 print("Создаем тестовое изображение...")  
 img = Image.new('L', (300, 300), color=255)  
 from PIL import ImageDraw  
  
 draw = ImageDraw.Draw(img)  
 draw.ellipse([75, 75, 225, 225], fill=50, outline=0)  
 draw.rectangle([120, 120, 180, 180], fill=200)  
 img.save(image\_path)  
 print(f"Тестовое изображение сохранено в {image\_path}")  
  
 *# ЭКСПЕРИМЕНТ 1: Масштабируемость* print("\n" + "#" \* 70)  
 print("# ЗАПУСК ЭКСПЕРИМЕНТА 1: МАСШТАБИРУЕМОСТЬ")  
 print("#" \* 70)  
  
 scalability\_results = experiment\_1\_scalability(image\_path, max\_processes=24)

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

*# ЭКСПЕРИМЕНТ 2: Сходимость* print("\n\n" + "#" \* 70)  
 print("# ЗАПУСК ЭКСПЕРИМЕНТА 2: СХОДИМОСТЬ")  
 print("#" \* 70)  
  
 convergence\_time, convergence\_algo = experiment\_2\_convergence(image\_path, n\_processes=24)  
  
 *# Итоговая сводка* print("\n\n" + "=" \* 70)  
 print("ИТОГОВАЯ СВОДКА ЭКСПЕРИМЕНТОВ")  
 print("=" \* 70)  
  
 print("\nЭксперимент 1 (Масштабируемость):")  
 print(f" - Итерации: 100,000")  
 print(f" - Диапазон процессов: 1-24")  
 t1 = scalability\_results[1]  
 t24 = scalability\_results[24]  
 speedup = t1 / t24  
 efficiency = (speedup / 24) \* 100  
 print(f" - Время на 1 процессе: {t1:.2f}с")  
 print(f" - Время на 24 процессах: {t24:.2f}с")  
 print(f" - Ускорение: {speedup:.2f}x")  
 print(f" - Эффективность: {efficiency:.1f}%")  
  
 *# Находим оптимальное количество процессов* best\_speedup = max(t1 / scalability\_results[n] for n in scalability\_results.keys())  
 best\_n\_proc = [n for n in scalability\_results.keys() if abs((t1 / scalability\_results[n]) - best\_speedup) < 0.01][0]  
 print(f" - Оптимальное число процессов: {best\_n\_proc} (ускорение {best\_speedup:.2f}x)")  
  
 print("\nЭксперимент 2 (Сходимость):")  
 print(f" - Итерации: 1,000,000")  
 print(f" - Процессы: 24")  
 print(f" - Время выполнения: {convergence\_time:.2f}с")  
 print(f" - Средняя скорость: {1000000 / convergence\_time:.0f} итераций/с")  
  
 initial\_metric = convergence\_algo.metric\_history[0]['metric']  
 final\_metric = convergence\_algo.metric\_history[-1]['metric']  
 improvement = ((initial\_metric - final\_metric) / initial\_metric) \* 100  
 print(f" - Начальная метрика: {initial\_metric:.4f}")  
 print(f" - Финальная метрика: {final\_metric:.4f}")  
 print(f" - Улучшение: {improvement:.1f}%")  
  
 print("\n" + "=" \* 70)  
 print("Все эксперименты завершены!")  
 print("=" \* 70)  
  
 print("\nСохраненные файлы:")  
 print(" - experiment1\_scalability.png - график масштабируемости")  
 print(" - experiment2\_convergence.png - график сходимости")  
 print(" - exp1\_progress\_\*proc.png - визуализация прогресса (эксперимент 1)")  
 print(" - exp2\_progress\_convergence.png - визуализация прогресса (эксперимент 2)")

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ А – ЛИСТИНГ КОДА

print("\n" + "=" \* 70)  
 print("АНАЛИЗ РЕЗУЛЬТАТОВ")  
 print("=" \* 70)  
  
 *# Анализ масштабируемости* print("\n1. Масштабируемость:")  
 speedups = [t1 / scalability\_results[n] for n in range(1, 25)]  
 efficiencies = [(speedups[n - 1] / n) \* 100 for n in range(1, 25)]  
  
 *# Находим когда эффективность падает ниже 50%* threshold\_50 = next((n for n in range(1, 25) if efficiencies[n - 1] < 50), 24)  
 print(f" - Эффективность >50% до {threshold\_50} процессов")  
  
 *# Средняя эффективность на разных диапазонах* avg\_eff\_small = np.mean(efficiencies[:4]) *# 1-4 процесса* avg\_eff\_medium = np.mean(efficiencies[4:8]) *# 5-8 процессов* avg\_eff\_large = np.mean(efficiencies[8:]) *# 9-24 процесса* print(f" - Средняя эффективность (1-4 процесса): {avg\_eff\_small:.1f}%")  
 print(f" - Средняя эффективность (5-8 процессов): {avg\_eff\_medium:.1f}%")  
 print(f" - Средняя эффективность (9-24 процесса): {avg\_eff\_large:.1f}%")  
  
 *# Анализ сходимости* print("\n2. Сходимость:")  
 metrics = [item['metric'] for item in convergence\_algo.metric\_history]  
 iterations\_conv = [item['iteration'] for item in convergence\_algo.metric\_history]  
  
 *# Скорость сходимости (изменение метрики на 100k итераций)* convergence\_rates = []  
 for i in range(1, len(metrics)):  
 iter\_diff = iterations\_conv[i] - iterations\_conv[i - 1]  
 metric\_diff = metrics[i - 1] - metrics[i]  
 rate = (metric\_diff / iter\_diff) \* 100000 *# на 100k итераций* convergence\_rates.append(rate)  
  
 print(f" - Начальная скорость сходимости: {convergence\_rates[0]:.4f} на 100k итераций")  
 print(f" - Конечная скорость сходимости: {convergence\_rates[-1]:.4f} на 100k итераций")  
  
 *# Когда достигается 80% улучшения* target\_metric = initial\_metric - 0.8 \* (initial\_metric - final\_metric)  
 iter\_80 = next((iterations\_conv[i] for i, m in enumerate(metrics) if m <= target\_metric), iterations\_conv[-1])  
 print(f" - 80% улучшения достигается на итерации: {iter\_80:,}")  
  
 print("\n" + "=" \* 70)

ПРИЛОЖЕНИЕ Б – ЭКСПЕРИМЕНТ 1

РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТА 1

Процессы Время (с) Ускорение Эффективность

----------------------------------------------------------------------

1 114.75 1.00x 100.0%

2 73.71 1.56x 77.8%

3 49.86 2.30x 76.7%

4 38.30 3.00x 74.9%

5 31.65 3.63x 72.5%

6 30.73 3.73x 62.2%

7 25.79 4.45x 63.6%

8 26.46 4.34x 54.2%

9 21.03 5.46x 60.6%

10 19.11 6.00x 60.0%

11 18.46 6.22x 56.5%

12 21.91 5.24x 43.6%

13 27.37 4.19x 32.2%

14 18.97 6.05x 43.2%

15 27.94 4.11x 27.4%

16 21.23 5.41x 33.8%

17 33.52 3.42x 20.1%

18 27.57 4.16x 23.1%

19 21.55 5.32x 28.0%

20 16.57 6.93x 34.6%

21 34.49 3.33x 15.8%

22 29.43 3.90x 17.7%

23 24.52 4.68x 20.3%

24 20.38 5.63x 23.5%

ПРИЛОЖЕНИЕ В – ЭКСПЕРИМЕНТ 2

РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТА 2

Итерация Метрика Улучшение от начала

----------------------------------------------------------------------

0 1000.0000 0.0%

5,000 1123.8553 -12.4%

10,000 820.7398 17.9%

15,000 641.8115 35.8%

20,000 526.8503 47.3%

25,000 474.9859 52.5%

30,000 420.6662 57.9%

35,000 374.1157 62.6%

40,000 336.3925 66.4%

45,000 320.2659 68.0%

50,000 309.9046 69.0%

100,000 184.3703 81.6%

150,000 128.2553 87.2%

200,000 93.9701 90.6%

250,000 69.1939 93.1%

300,000 63.2771 93.7%

350,000 52.9486 94.7%

400,000 44.7357 95.5%

450,000 44.5390 95.5%

500,000 43.8196 95.6%

550,000 38.9934 96.1%

ПРОДОЛЖЕНИЕ ПРИЛОЖЕНИЯ В

600,000 34.6857 96.5%

650,000 30.3647 97.0%

700,000 24.0745 97.6%

750,000 28.9445 97.1%

800,000 33.6119 96.6%

850,000 27.3215 97.3%

900,000 22.6099 97.7%

950,000 21.2019 97.9%

1,000,000 22.1736 97.8%

ПРИЛОЖЕНИЕ Г – РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ

ИТОГОВАЯ СВОДКА ЭКСПЕРИМЕНТОВ

Эксперимент 1 (Масштабируемость):

- Итерации: 100,000

- Диапазон процессов: 1-24

- Время на 1 процессе: 114.75с

- Время на 24 процессах: 20.38с

- Ускорение: 5.63x

- Эффективность: 23.5%

- Оптимальное число процессов: 20 (ускорение 6.93x)

Эксперимент 2 (Сходимость):

- Итерации: 1,000,000

- Процессы: 24

- Время выполнения: 179.40с

- Средняя скорость: 5574 итераций/с

- Начальная метрика: 1000.0000

- Финальная метрика: 22.1736

- Улучшение: 97.8%