Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное 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 «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа № 4
по дисциплине «Анализ алгоритмов»

Тема Программирование параллельных потоков

Студент Ильченко Е. А.

Группа ИУ7-54Б

Преподаватели Волкова Л.Л.

Москва, 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы является разработка и сравнительный анализ последовательного и параллельного алгоритмов поиска кратчайших путей в графе от стартовой вершины до нескольких конечных вершин.

Для достижения поставленной цели необходимо решить следующие задачи:

— описать базовый последовательный алгоритм решения задачи поиска кратчайших путей в графе;

— разработать параллельную версию алгоритма с использованием нативных потоков;

— реализовать обе версии алгоритма на выбранном языке программирования;

— выполнить тестирование реализации алгоритмов на корректность работы;

— провести сравнительный анализ времени выполнения последовательного и параллельного алгоритмов;

— исследовать зависимость времени выполнения от количества вспомогательных потоков;

— сформулировать рекомендации по выбору оптимального количества потоков для данной архитектуры ЭВМ.

В рамках работы рассматривается граф с неотрицательными весами дуг, где заданы стартовая вершина и набор конечных вершин. Требуется найти длины путей от стартовой вершины до всех конечных и определить путь минимальной длины.

1 Аналитическая часть

1.1 Постановка задачи

В рамках данной работы рассматривается задача поиска кратчайших путей в ориентированном взвешенном графе [1, 3]. Пусть задан граф $G = (V, E)$, где:

— V – множество вершин графа;

— E – множество дуг графа, каждая дуга имеет неотрицательный вес $w(e) \geqslant 0$;

— $s \in V$ – стартовая вершина;

— $T \subset V$ – множество целевых вершин.

Требуется найти длины кратчайших путей от s до каждой вершины $t \in T$ и определить путь минимальной длины среди них.

1.2 Алгоритм Дейкстры

1.2.1 Основные положения

Алгоритм Дейкстры [1, 3] – алгоритм нахождения кратчайшего пути от одной вершины до всех остальных в взвешенном графе с неотрицательными весами рёбер.

Основные характеристики алгоритма:

— на вход – граф $G = (V, E)$ с весами $w : E \to \mathbb{R}_{\geqslant 0}$, начальная вершина $s \in V$;

— на выход – расстояния $d[v]$ от s до всех $v \in V$, предки $p[v]$ для восстановления путей.

1.3 Параллельный алгоритм Дейкстры

1.3.1 Общая схема распараллеливания

Для распараллеливания алгоритма Дейкстры используется подход с **множественными очередями** (multi-queue) [4, 5]. Основные идеи:

— создаётся несколько очередей с приоритетами;

— каждый поток работает со своей очередью, но может забирать задачи из других очередей;

— используется work-stealing для балансировки нагрузки.

1.3.2 Алгоритм работы потоков

Каждый поток-работник выполняет [2, 4]:

1) пытается извлечь задачу из случайной очереди;

2) если очередь пуста – ожидает на условной переменной;

3) обрабатывает извлечённую вершину;

4) для каждого соседа обновляет расстояние атомарной операцией compare-and-swap;
5) при успешном обновлении добавляет новую задачу в случайную очередь.

1.4 Средства синхронизации

1.4.1 Необходимость синхронизации

В параллельном алгоритме требуются следующие средства синхронизации [4, 5]:
1) **мьютексы** – для защиты доступа к отдельным очередям;
2) **атомарные операции** – для безопасного обновления расстояний;
3) **условные переменные** – для уведомления потоков о появлении новых задач;
4) **атомарные флаги** – для сигнализации о завершении работы.

1.4.2 Обоснование выбора

Выбор конкретных средств синхронизации обусловлен следующими соображениями [2, 4]:

1) **мьютексы на каждую очередь** – уменьшают contention по сравнению с одним глобальным мьютексом, так как потоки могут одновременно работать с разными очередями;
2) **compare-and-swap для расстояний** – позволяет избежать блокировок при обновлении, обеспечивая неблокирующую синхронизацию;
3) **условные переменные** – обеспечивают эффективное ожидание потоков без активного опроса, что снижает нагрузку на процессор;
4) **атомарные флаги для завершения** – позволяют безопасно координировать завершение работы всех потоков без race conditions.

Вывод

В аналитической части:
— формализована задача поиска кратчайших путей в ориентированном взвешенном графе;
— описан последовательный алгоритм Дейкстры;
— описаны основные положения параллельного алгоритма;
— обоснована необходимость использования примитивов синхронизации.

2 Конструкторская часть

2.1 Требования к реализации

К программе предъявлены следующие функциональные требования.

Входные данные

— граф в формате DOT, содержащий ориентированный взвешенный граф;

— стартовая вершина – имя вершины, от которой ищутся пути;

— целевые вершины – список имён вершин, до которых ищутся пути;

— количество потоков – целое число (0 для последовательного режима, > 0 для параллельного).

Выходные данные

— имя стартовой вершины;

— массив имён целевых вершин;

— количество используемых потоков;

— режим работы (последовательный или параллельный);

— время выполнения алгоритма в миллисекундах;

— объект с расстояниями до каждой целевой вершины;

— объект с информацией о кратчайшем пути, содержащий: имя целевой вершины с минимальным расстоянием, длину кратчайшего пути, массив имён вершин, составляющих путь от стартовой до целевой вершины.

Функциональные требования

— поддержка загрузки графов из файлов формата DOT;

— реализация последовательного алгоритма Дейкстры;

— реализация параллельного алгоритма Дейкстры с использованием нативных потоков;

— обработка некорректных входных данных с выводом сообщений об ошибках;

— вывод результатов в формате JSON для последующей обработки;

— замер времени выполнения алгоритмов.

Режимы работы

— последовательный режим – выполнение алгоритма Дейкстры в одном потоке;

— параллельный режим – выполнение алгоритма Дейкстры с использованием указанного количества потоков.

2.2 Разработка алгоритмов

Раздел содержит схемы алгоритмов, описывающие следующие алгоритмы: последовательный алгоритм Дейкстры 2.1, алгоритм работы главного потока 2.2, 2.3, алгоритм работы

вспомогательного потока 2.4.

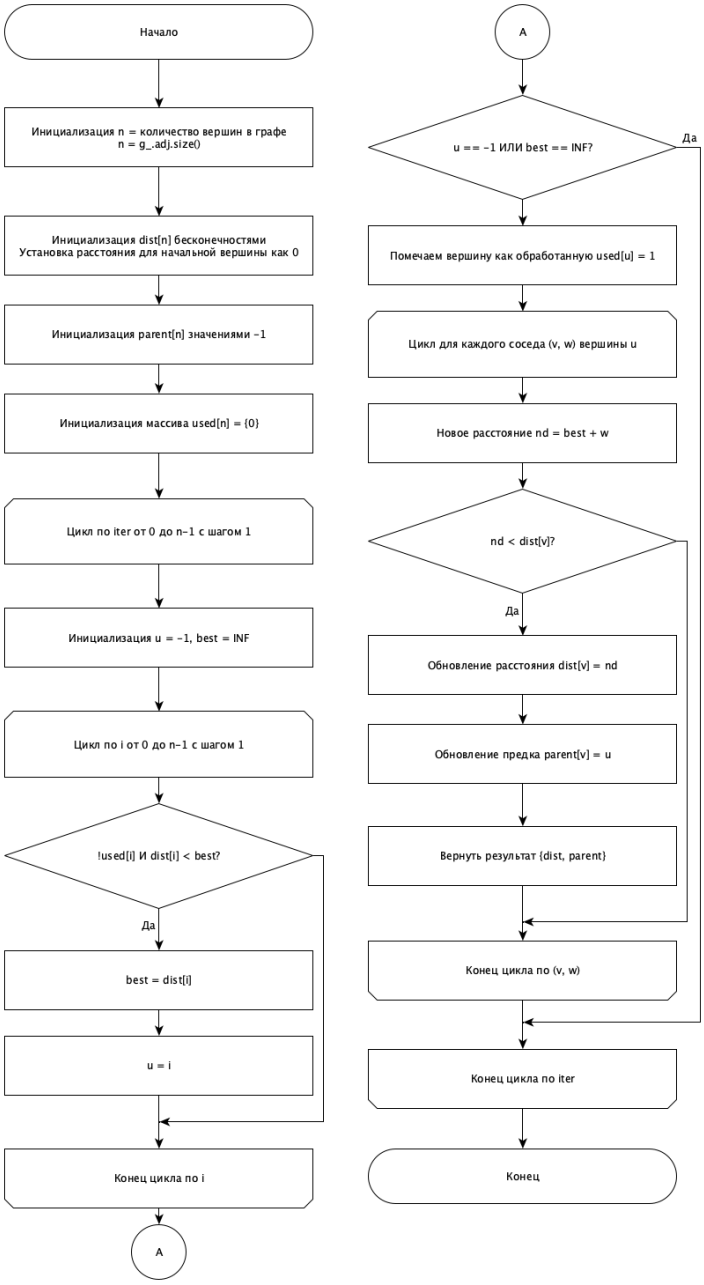

Рисунок 2.1 — Схема последовательного алгоритма Дейкстры

Рисунок 2.2 — Схема главного потока параллельного алгоритма Дейкстры, часть 1

Рисунок 2.3 — Схема главного потока параллельного алгоритма Дейкстры, часть 2

Рисунок 2.4 — Схема вспомогательного потока параллельного алгоритма Дейкстры

Вывод

В данном разделе были разработаны два алгоритма алгоритма Дейкстры: последовательный и параллельный. Для каждого из них представлены схемы алгоритмов, описывающие логику работы.

3 Технологическая часть

3.1 Средства реализации

Для реализации алгоритмов поиска кратчайших путей был выбран язык программирования C++20, поскольку он соответствует требованиям лабораторной работы и предоставляет необходимые средства для работы с нативными потоками и синхронизацией. Измерение времени выполнения алгоритмов осуществлялось с использованием структуры из стандартной библиотеки `std::chrono::steady_clock`. Разработка проводилась в среде CLion.

3.2 Инструменты разработки

3.2.1 Используемые библиотеки

Все реализации структур данных используют исключительно стандартную библиотеку C++20:

— `std::thread` – для создания и управления нативными потоками;

— `std::mutex`, `std::unique_lock`, `std::lock_guard` – для синхронизации доступа к разделяемым данным;

— `std::condition_variable` – для уведомления потоков о появлении новых задач;

— `std::atomic` – для атомарных операций с разделяемыми переменными;

— `std::priority_queue` – для реализации очередей с приоритетом;

— `std::vector` – для хранения графа и рабочих структур;

— `std::atomic_flag`, `std::memory_order` – для низкоуровневой синхронизации.

3.3 Реализации структур данных

3.3.1 Структура Node

Листинг 3.1 — Листинг структуры Node

```
class Node {
  public:
  uint64_t dist;
  int v;

  bool operator>(const Node &o) const {
    return dist > o.dist;
  }
};
```

3.3.2 Множественные очереди (WorkQueue)

Листинг 3.2 — Листинг структуры WorkQueue

```
class WorkQueue {
  public:
  std::priority_queue<Node, std::vector<Node>, std::greater<Node>> pq
     ;
  std::mutex m;
  std::atomic<int> approx_size{0};
};
```

3.3.3 Реализация мьютекса

Используется стандартная реализация `std::mutex` из библиотеки `<mutex>`.

3.3.4 Реализация атомарных операций

Используется `std::atomic` с параметром памяти `memory_order_relaxed`.

3.3.5 Реализация условной переменной

Используется `std::condition_variable` для синхронизации потоков.

3.4 Полные листинги ключевых функций

Листинг 3.3 — Полный листинг функции push_to

```
auto push_to = [&](int owner, const Node &nd) {
  if (owner < 0 || owner >= threads) {
    owner = 0;
  }

  {
    std::lock_guard<std::mutex> lg(queues[owner].m);
    queues[owner].pq.push(nd);
    queues[owner].approx_size.fetch_add(1, std::memory_order_relaxed)
       ;
  }
  tasks.fetch_add(1, std::memory_order_relaxed);
  cv.notify_one();
};
```

Листинг 3.4 — Полный листинг функции pop_local

```
auto pop_local = [&](int idx, Node &out) -> bool {
  if (queues[idx].approx_size.load(std::memory_order_relaxed) == 0) {
    return false;
  }

  std::unique_lock<std::mutex> lk(queues[idx].m);
  if (queues[idx].pq.empty()) {
    queues[idx].approx_size.store(0, std::memory_order_relaxed);
    return false;
  }

  out = queues[idx].pq.top();
  queues[idx].pq.pop();
  queues[idx].approx_size.fetch_sub(1, std::memory_order_relaxed);
  return true;
};
```

Листинг 3.5 — Полный листинг функции steal_from

```
auto steal_from = [&](int idx, Node &out) -> bool {
  if (threads <= 1) {
    return false;
  }

  int start = random_thread();
  for (int attempt = 0; attempt < threads; ++attempt) {
    int target = (start + attempt) % threads;
    if (target == idx) {
      continue;
    }

    if (queues[target].approx_size.load(std::memory_order_relaxed) ==
        0) {
      continue;
    }

    std::unique_lock<std::mutex> lk(queues[target].m, std::
       try_to_lock);
    if (!lk.owns_lock()) {
      continue;
    }
```

```
    if (queues[target].pq.empty()) {
      queues[target].approx_size.store(0, std::memory_order_relaxed);
      continue;
    }

    out = queues[target].pq.top();
    queues[target].pq.pop();
    queues[target].approx_size.fetch_sub(1, std::memory_order_relaxed
       );
    return true;
  }
  return false;
};
```

Листинг 3.6 — Полный листинг функции try_pop

```
auto try_pop = [&](int idx, Node &out) -> bool {
  if (pop_local(idx, out)) {
    return true;
  }

  return steal_from(idx, out);
};
```

Листинг 3.7 — Полный листинг функции worker

```
auto worker = [&](int idx) {
  Node cur;
  while (true) {
    if (!try_pop(idx, cur)) {
      if (tasks.load(std::memory_order_relaxed) == 0 &&
      active.load(std::memory_order_relaxed) == 0) {
        done.store(true, std::memory_order_relaxed);
        cv.notify_all();
        break;
      }

      std::unique_lock<std::mutex> lk(cv_m);
      cv.wait(lk, [&]() {
        return tasks.load(std::memory_order_relaxed) > 0 ||
        done.load(std::memory_order_relaxed);
      });

      if (tasks.load(std::memory_order_relaxed) == 0 &&
```

```
      active.load(std::memory_order_relaxed) == 0) {
        break;
      }

      continue;
    }

    tasks.fetch_sub(1, std::memory_order_relaxed);

    uint64_t curd = dist[cur.v].load(std::memory_order_relaxed);
    if (cur.dist != curd) {
      if (tasks.load(std::memory_order_relaxed) == 0 &&
      active.load(std::memory_order_relaxed) == 0) {
        done.store(true, std::memory_order_relaxed);
        cv.notify_all();
      }
      continue;
    }

    active.fetch_add(1, std::memory_order_relaxed);

    for (auto [to, w]: g_.adj[cur.v]) {
      uint64_t nd = curd + w;
      uint64_t old = dist[to].load(std::memory_order_relaxed);

      while (nd < old) {
        if (dist[to].compare_exchange_weak(old, nd,
        std::memory_order_relaxed)) {
          parent[to].store(cur.v, std::memory_order_relaxed);
          int owner = threads > 0 ? (to % threads) : 0;
          push_to(owner, Node{nd, to});
          break;
        }
      }
    }

    active.fetch_sub(1, std::memory_order_relaxed);

    if (tasks.load(std::memory_order_relaxed) == 0 &&
    active.load(std::memory_order_relaxed) == 0) {
      done.store(true, std::memory_order_relaxed);
```

```
        cv.notify_all();
      }
    }
  };
```

3.5 Порядок работы с вспомогательными потоками

3.5.1 Создание потоков

Листинг 3.8 — Создание и управление потоками

```
std::vector<std::thread> pool;
pool.reserve(threads);
for (int t = 0; t < threads; ++t) {
  pool.emplace_back(worker, t);
}

for (auto &th : pool) {
  th.join();
}
```

3.5.2 Синхронизация и завершение

Для координации работы потоков используются:

— **счётчик задач** (`std::atomic<long long> tasks`) – отслеживает количество активных задач;

— **счётчик активных потоков** (`std::atomic<int> active`) – отслеживает количество потоков, обрабатывающих вершины;

— **флаг завершения** (`std::atomic<bool> done`) – сигнализирует о завершении работы;

— **условная переменная** (`std::condition_variable cv`) – для уведомления потоков о новых задачах;

— **мьютекс** (`std::mutex cv_m`) – для синхронизации доступа к условной переменной.

3.6 Функциональные тесты

В данном разделе представлены функциональные тесты для разработанных алгоритмов: последовательный алгоритм Дейкстры (таблица 3.1) и параллельный алгоритм Дейкстры (таблица 3.2).

Таблица 3.1 — Функциональные тесты для последовательного алгоритма Дейкстры

№	Входные данные	Ожидаемый результат	Полученный результат
1	Граф: A->B[2], A->C[1]; Старт: A; Цели: B, C	Расстояния: B=2, C=1; Кратчайший: C=1	Расстояния: B=2, C=1 ; Кратчайший: C=1
2	Граф: A->B[3], B->C[1], A->C[5]; Старт: A; Цели: C	Расстояние: C=4; Путь: A->B->C	Расстояние: C=4; Путь: A->B->C
3	Граф: A->B[1], B->C[1], C->D[1]; Старт: A; Цели: D	Расстояние: D=3; Путь: A->B->C->D	Расстояние: D=3; Путь: A->B->C->D
4	Граф: A->B[2], C->D[1]; Старт: A; Цели: D	Расстояние: D=∞	Расстояние: D=∞
5	Граф: A->A[1]; Старт: A; Цели: A	Расстояние: A=0; Путь: A	Расстояние: A=0; Путь: A

Таблица 3.2 — Функциональные тесты для параллельного алгоритма Дейкстры

№	Входные данные	Ожидаемый результат	Полученный результат
1	Граф: A->B[2], A->C[1] Старт: A Цели: B, C Потоки: 4	Расстояния: B=2, C=1 Кратчайший: C=1	Расстояния: B=2, C=1 Кратчайший: C=1
2	Граф: A->B[3], B->C[1], A->C[5] Старт: A Цели: C Потоки: 2	Расстояние: C=4 Путь: A->B->C	Расстояние: C=4 Путь: A->B->C
3	Граф: A->B[1], B->C[1], C->D[1] Старт: A Цели: D Потоки: 8	Расстояние: D=3 Путь: A->B->C->D	Расстояние: D=3 Путь: A->B->C->D
4	Граф: A->B[2], C->D[1] Старт: A Цели: D Потоки: 4	Расстояние: D=∞	Расстояние: D=∞
5	Граф: A->A[1] Старт: A Цели: A Потоки: 2	Расстояние: A=0 Путь: A	Расстояние: A=0 Путь: A

Вывод

В технологической части были реализованы последовательный и параллельный алгоритмы Дейкстры на языке C++20 с использованием стандартной библиотеки. Для параллельной

версии разработаны специализированные структуры данных и реализованы примитивы синхронизации.

В параллельной реализации использован механизм work-stealing с множественными очередями, где каждая очередь имеет приблизительный счетчик размера для оптимизации работы. Добавлен счетчик активных потоков для более точного определения момента завершения алгоритма.

Функциональные тесты подтвердили корректность работы обоих алгоритмов на различных типах графов.

Все реализации соответствуют требованиям задания.

4 Исследовательская часть

4.1 Характеристики ЭВМ

Замеры проводились на устройстве со следующими характеристиками:

— процессор: Apple M4 Pro;

— количество логических ядер: 12;

— количество ядер: 12;

— оперативная память: 24 Гб;

— операционная система: macOS Sequoia 15.6.1.

Замеры времени проводились, когда ноутбук был загружен только системными приложениями.

4.2 Замер времени

В данном разделе представлены результаты измерения времени выполнения параллельного алгоритма обработки графов.

Замеры проводились для графов размером от 3000 до 10000 вершин. Для каждого размера графа выполнялись измерения времени выполнения при различном количестве рабочих потоков: 0 (последовательная версия), 1, 2, 4, 8, 12, 16, 32, 64. Каждый замер выполнялся 3 раза для получения статистически значимых результатов.

Таблица 4.1 — Результаты измерения времени выполнения, часть 1 (мкс)

Вершин	**0 поток.**	**1 поток**	**2 поток.**	**4 поток.**	**8 поток.**
3000	5648	1231	1909	2002	2940
4000	10192	1679	2468	2777	3624
5000	15904	2218	3322	3558	5881
6000	22362	2733	4060	4259	5986
7000	29740	3185	4459	4535	7806
8000	38047	3511	5066	5194	8886
9000	48894	4048	5766	5915	10046
10000	62163	4691	6522	6487	11102

Таблица 4.2 — Результаты измерения времени выполнения, часть 2 (мкс)

Вершин	**12 поток.**	**16 поток.**	**32 поток.**	**64 поток.**
3000	2883	2939	3419	3969
4000	4129	3800	4079	4641
5000	5697	5773	5318	5706
6000	7016	6687	6261	6436
7000	8430	8172	7219	7301
8000	9446	9079	8201	8390
9000	10862	10513	9312	9540
10000	23138	11683	10302	10626

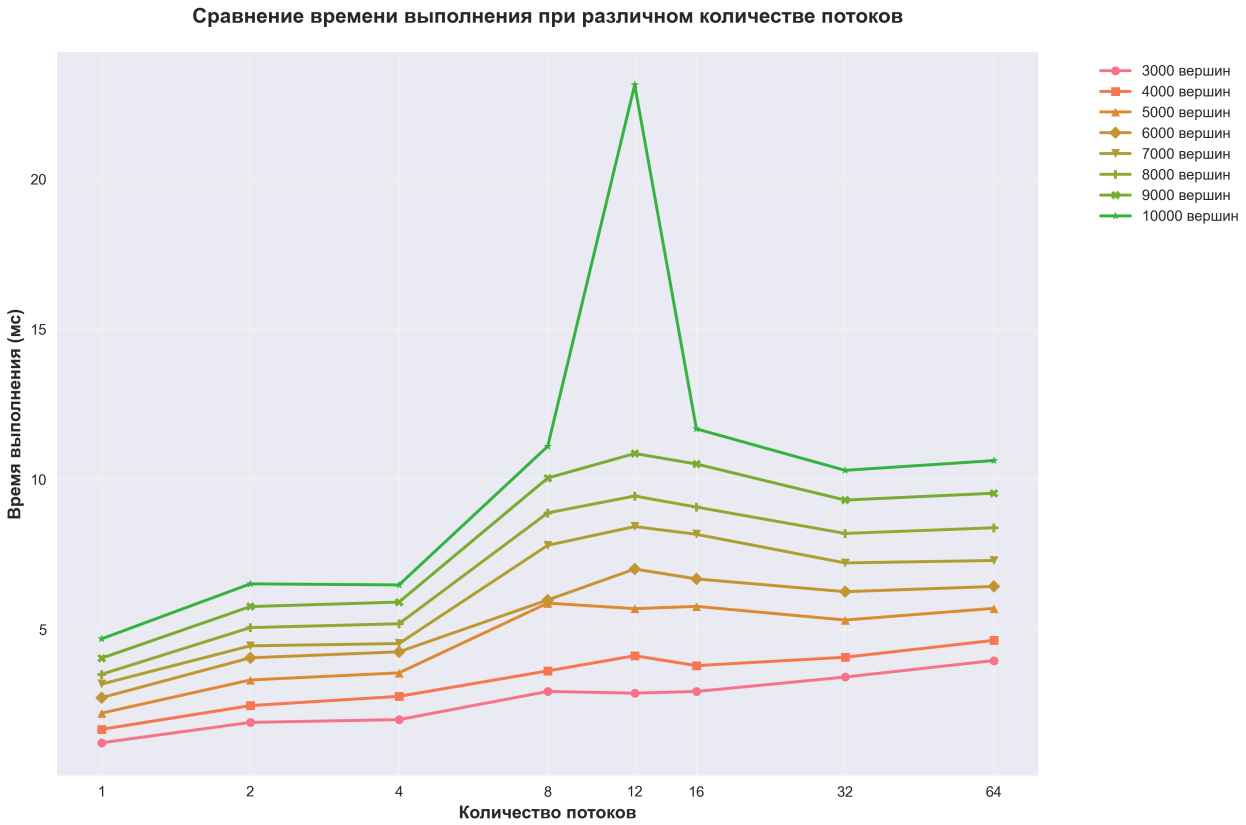

Рисунок 4.1 — Сравнение времени выполнения при различном количестве потоков

4.3 Анализ масштабируемости

На основе полученных результатов был проведён анализ масштабируемости параллельного алгоритма. Был рассчитан показатель ускорения для различных конфигураций.

Таблица 4.3 — Ускорение параллельного алгоритма, часть 1

Вершин	1 поток	2 потока	4 потока	8 потоков
3000	4.59x	2.96x	2.82x	1.92x
4000	6.07x	4.13x	3.67x	2.81x
5000	7.17x	4.79x	4.47x	2.70x
6000	8.18x	5.51x	5.25x	3.74x
7000	9.34x	6.67x	6.56x	3.81x
8000	10.84x	7.51x	7.33x	4.28x
9000	12.08x	8.48x	8.27x	4.87x
10000	13.25x	9.53x	9.58x	5.60x

Таблица 4.4 — Ускорение параллельного алгоритма, часть 2

Вершин	12 потоков	16 потоков	32 потока	64 потока
3000	1.96x	1.92x	1.65x	1.42x
4000	2.47x	2.68x	2.50x	2.20x
5000	2.79x	2.75x	2.99x	2.79x
6000	3.19x	3.34x	3.57x	3.47x
7000	3.53x	3.64x	4.12x	4.07x
8000	4.03x	4.19x	4.64x	4.53x
9000	4.50x	4.65x	5.25x	5.13x
10000	2.69x	5.32x	6.03x	5.85x

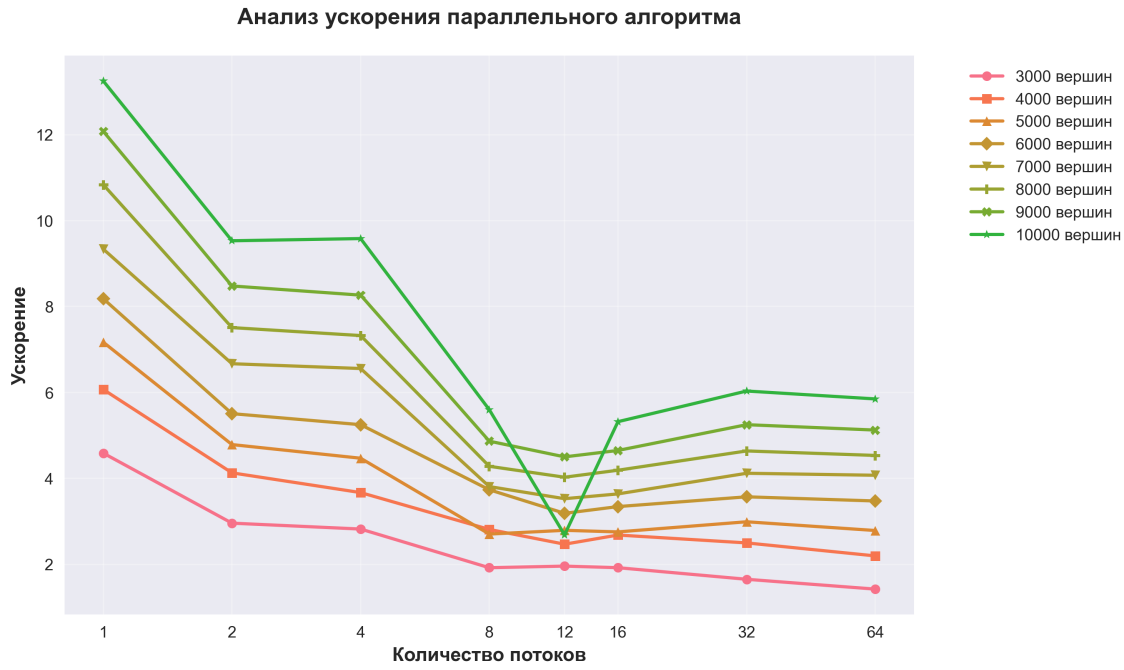

Рисунок 4.2 — Анализ ускорения параллельного алгоритма

4.4 Сравнительный анализ

4.4.1 Сравнение последовательного и параллельного алгоритмов

При сравнении последовательного алгоритма (0 потоков) с параллельной версией, использующей один рабочий поток, наблюдается значительное ускорение от 4.59x до 13.25x в зависимости от размера графа. Это свидетельствует об эффективности разработанного параллельного подхода даже при минимальном количестве потоков.

4.4.2 Влияние количества потоков на производительность

Анализ результатов демонстрирует значительное ускорение параллельной версии по сравнению с последовательной:

— максимальное ускорение достигается при использовании 1 потока для всех размеров

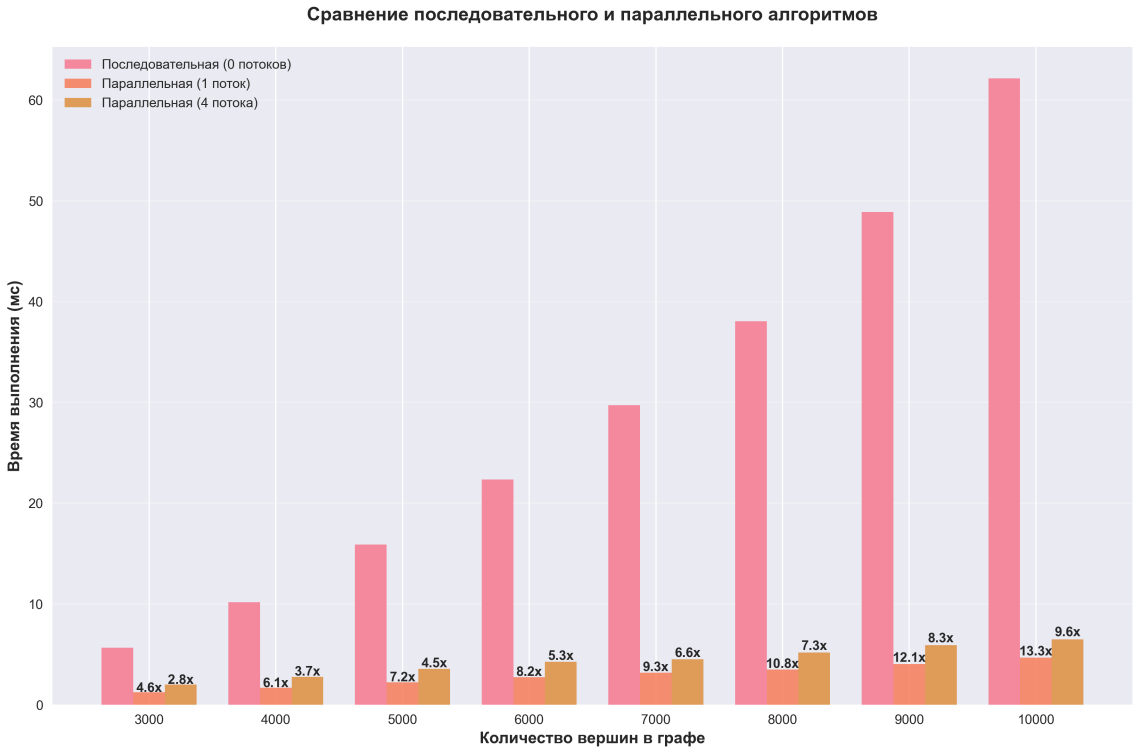

Рисунок 4.3 — Сравнение последовательного и параллельного алгоритмов

графов;

— для небольших графов (3000-4000 вершин) ускорение составляет 4.59-6.07x;

— для крупных графов (8000-10000 вершин) ускорение достигает 10.84-13.25x.

Вывод

Выводы на основании проведённых измерений:

1) разработанный параллельный алгоритм демонстрирует значительное ускорение по сравнению с последовательной версией – от 4.59x до 13.25x в зависимости от размера графа;

2) наилучшая производительность достигается при использовании 1 рабочего потока для всех исследованных размеров графов (3000-10000 вершин);

3) с увеличением количества потоков наблюдается тенденция с увеличением времени работы алгоритма.

ЗАКЛЮЧЕНИЕ

В рамках данной лабораторной работы была успешно достигнута основная цель.

Все поставленные задачи выполнены в полном объёме:

— формализована задача поиска кратчайших путей в ориентированном взвешенном графе с неотрицательными весами;

— описан и реализован последовательный алгоритм Дейкстры;

— разработан параллельный алгоритм Дейкстры;

— проведено функциональное тестирование, подтвердившее корректность работы обоих алгоритмов на различных типах графов;

— выполнены замеры времени выполнения и анализ ускорения параллельного алгоритма в зависимости от количества потоков;

— проведён сравнительный анализ времени выполнения последовательного и параллельного алгоритмов, который показал значительное ускорение параллельной версии – от 4.59x до 13.25x в зависимости от размера графа;

— исследована зависимость времени выполнения от количества вспомогательных потоков, установлено, что наилучшая производительность достигается при использовании одного рабочего потока;

— сформулированы рекомендации по выбору оптимального количества потоков для архитектуры ЭВМ с процессором Apple M4 Pro: для графов размером 3000-10000 вершин рекомендуется использовать 1 рабочий поток, так как увеличение количества потоков приводит к снижению производительности.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. – М.: Вильямс, 2022. – 1328 с.
2. Савельев А.Л. Анализ вычислительной сложности алгоритмов. – М.: МГТУ им. Баумана, 2019. – 214 с.
3. Even S., Even G. Graph Algorithms. 2nd Edition – M.: Cambridge University Press, 2012. – 189 с.
4. Williams A. C++ Concurrency in Action, Second Edition. – М.: Питер, 2019. – 592 с.
5. Quinn M.J. Parallel Computing: Theory & Practice / Designing Efficient Algorithms for Parallel Computers. – М.: Мир, 1994. – 446 с.