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 «Программное обеспечение ЭВМ и информационные технологии»

НАПРАВЛЕНИЕ ПОДГОТОВКИ «09.03.04 Программная инженерия»

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №4

Название: Параллельные вычисления. Алгоритм волновой трассировки

Дисциплина: Анализ алгоритмов

Студент	ИУ7-56Б		Т. А. Казаева
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель			Л. Л. Волкова
		Подпись, дата	И. О. Фамилия

Москва, 2021 г.

Содержание

Страница

Введение **2**

1 Аналитический раздел **3**
1.1 Алгоритм волновой трассировки 3
1.2 Поиск в ширину 4
1.3 Вывод 4

2 Конструкторский раздел **5**
2.1 Схемы алгоритмов 5
Вывод 9

3 Технологический раздел **10**
3.1 Требования к ПО 10
3.2 Средства реализации 10
3.3 Листинги кода 10
3.4 Тестирование ПО 15
3.5 Вывод 15

4 Исследовательский раздел **16**
4.1 Технические характеристики 16
4.2 Описание системы тестирования 16
4.3 Таблица времени выполнения алгоритмов 16
4.4 Графики функций 17
4.5 Вывод 18

Заключение **19**

Список литературы **20**

Введение

Алгоритм нахождения кратчайшего пути в лабиринте используется во множестве разных областей, начиная от компьютерных игр и заканчивая трассировкой печатных плат, соединительных проводников и поверхностей микросхем. Хоть алгоритм и прост в реализации, вычислительные расходы на его исполнение достаточно велики.

В представленной работе исследуется реализация алгоритма Ли на параллельных процессах. Для реализации поставленной задачи следует выполнить следующие этапы:

- Проанализировать алгоритм волновой трассировки;
- Выделить стратегии распараллеливания алгоритма;
- Разработать ПО для решения поставленной задачи;
- Сравнить результаты работы последовательного и параллельного алгоритмов с помощью реализованного ПО.

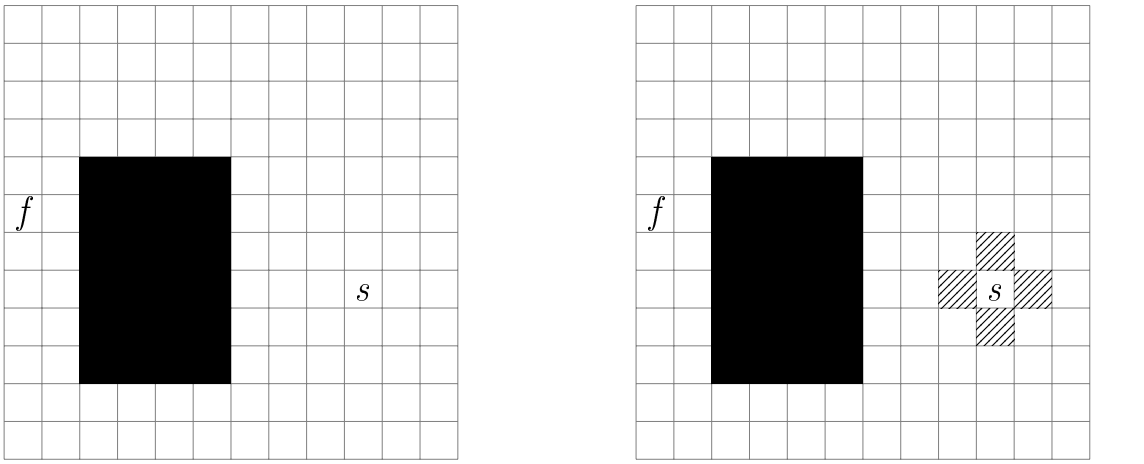
Также в данной работе следует ответить на вопрос: "Всегда ли при увеличении количества потоков прирост времени выполнения программы дает выигрыш вдвое?"

Результаты сравнительного анализа будут приведены в виде таблиц и графиков, из чего можно будет сделать вывод об эффективности оптимизаций, предложенных в данной работе и ответить на поставленный вопрос.

1. Аналитический раздел

1.1 Алгоритм волновой трассировки

В своей работе Ли [1] предлагал решать задачу о нахождении минимального пути в лабиринте путем представления пространства в виде матрицы. Каждая ячейка матрицы либо является препятствием, либо проходом. Рисунок 1.1 демонстрирует принцип работы алгоритма. Здесь разными цветами отмечены ячейки с разными свойствами – на рисунке 1.1а. белым цветом отмечена ячейка – проход, черным – препятствие.



(а.) Начальная конфигурация (б.) Распространение волны

Рисунок 1.1 – Матрица волнового алгоритма Ли

Начальная клетка осуществляет отправку сообщения четырем соседним. Сообщение представляет собой волну, продемонстрированную на рисунке 1.1б., Каждая захваченная волной ячейка, в свою очередь, отправляет сообщение четырем соседним. Сообщение представляет собой число, равное кратчайшему Манхэттенскому расстоянию от стартовой ячейки до ячейки – получателя.

Алгоритм имеет три стадии – инициализация, распространение волны и восстановление пути, вторая стадия – самая затратная с точки зрения вычислительных мощностей. Последняя стадия в рамках данной работы интереса не представляет, поэтому она будут опущена.

1.2 Поиск в ширину

Одно из возможных решений алгоритма Ли основано на поиске в ширину[2]. Существуют два варианта обхода графа при поиске в ширину: прямой обход графа («top-down») и обратный обход графа («bottom-up»). В рассматриваемом алгоритме используется прямой обход, который предполагает, что активные вершины помечают ближайшие соседние. В случае карты - лабиринта соседними вершинами являются верняя, нижняя, правая и левая ячейки карты, если они являются проходом, а не препятствием. Алгоритм поиска в ширину реализуется с помощью очередей. Применяя его к задаче нахождения кратчайшего пути в лабиринте, можно сформировать следующее формальное описание:

(i) Создать пустую очередь;
(ii) Поместить в очередь начальный узел, отметив его числом 0;
(iii) Извлечь из начала очереди ячейку s;
(iv) Если координаты ячейки s совпадают с целевыми, то вернуть метку этой ячейки и завершить процедуру;
(v) Иначе, для каждой из четырех соседних клеток распространить волну (присвоить метку, большую метки рассматриваемой ячейки на 1);
(vi) Если очередь пуста, то все ячейки были просмотрены, следовательно, целевая ячейка недостижима из начальной; завершить поиск.
(vii) Вернуться к пункту (iii).

1.3 Вывод

Реализация второго этапа алгоритма Ли (распространение волны) сводится к решению задачи поиска в ширину на графе, представленном в качестве карты обхода и её распараллеливанию. В целях упрощения задачи, этап восстановления пути будет опущен – в качетсве результата достаточно предоставить найденное расстояние, иначе говоря, метку искомой вершины графа.

2. Конструкторский раздел

2.1 Схемы алгоритмов

Обращение к элементам массивов на схеме алгоритмов обозначено подстрочными индексами. Операция присваивания обозначена как "← операции "равно"и "не равно"как "="и "#"соответственно. На рисунке 2.1 представлены схемы параллельного и синхронного выполнения алгоритма Ли. На рисунке 2.2 представлена схема поиска в ширину. На рисунке 2.3 представлена схема распространения волны.

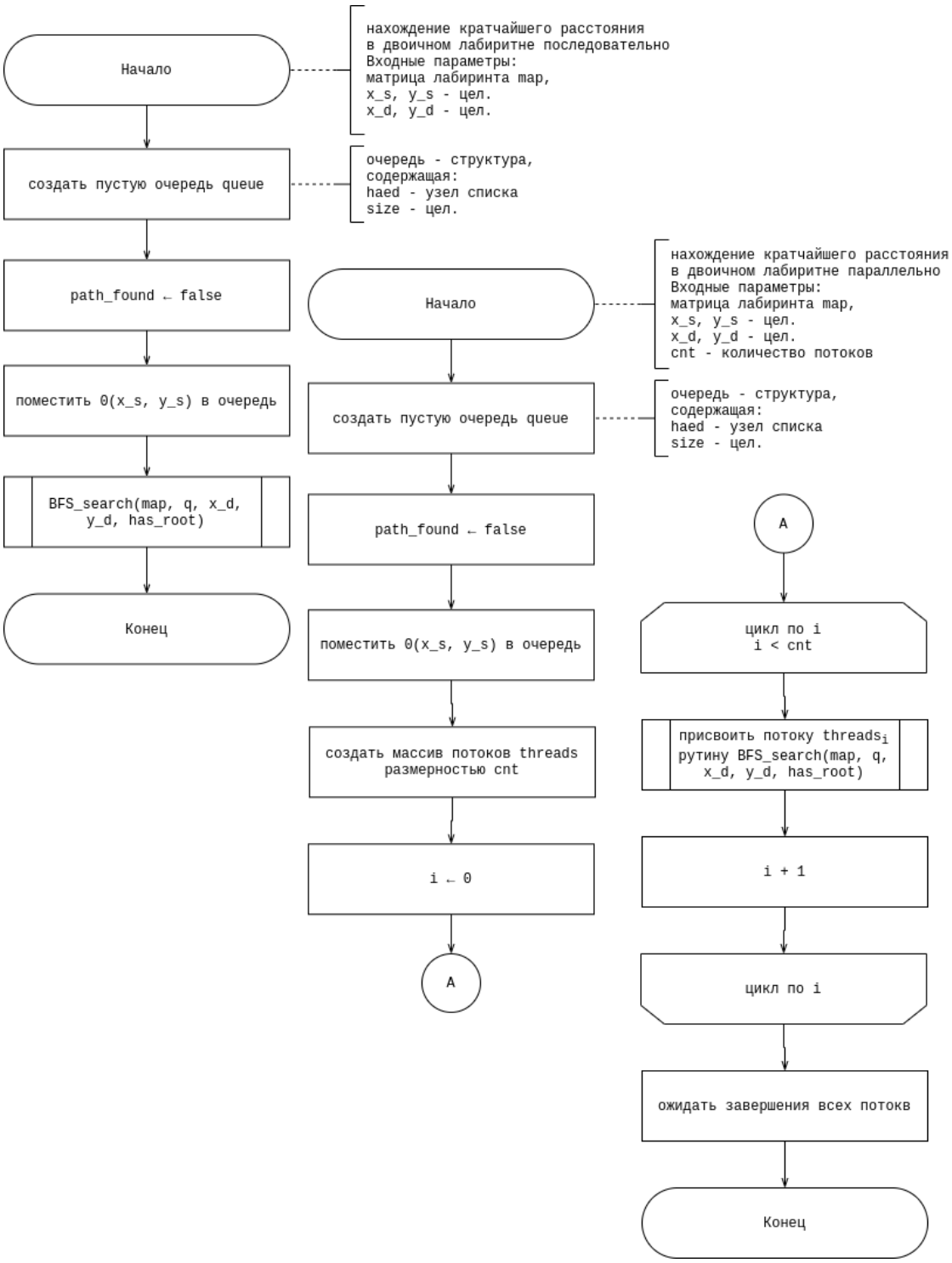


Рисунок 2.1 – Схемы синхронного и параллельного алгоритмов Ли

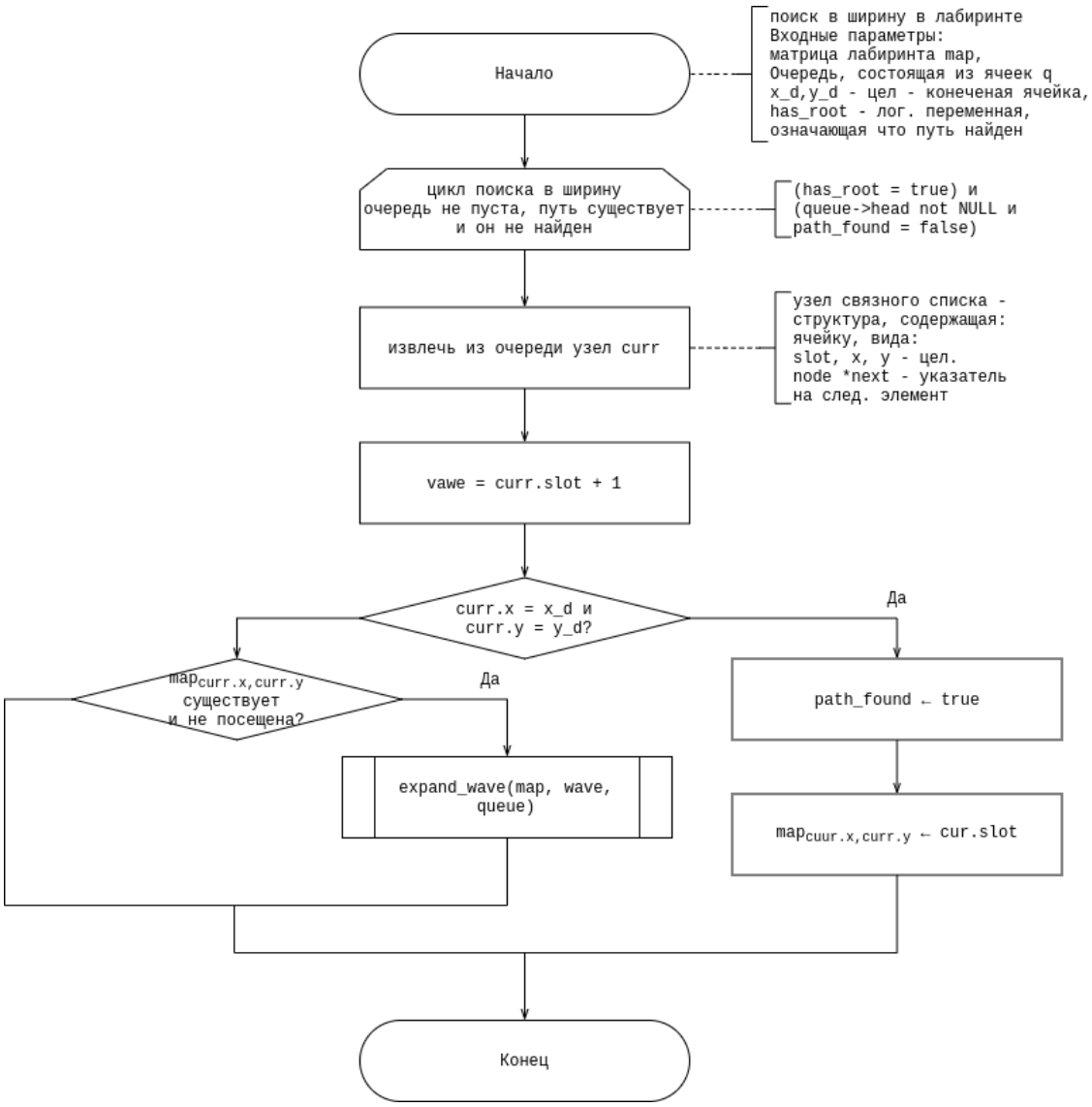


Рисунок 2.2 – Схема поиска в ширину на графе - матрице

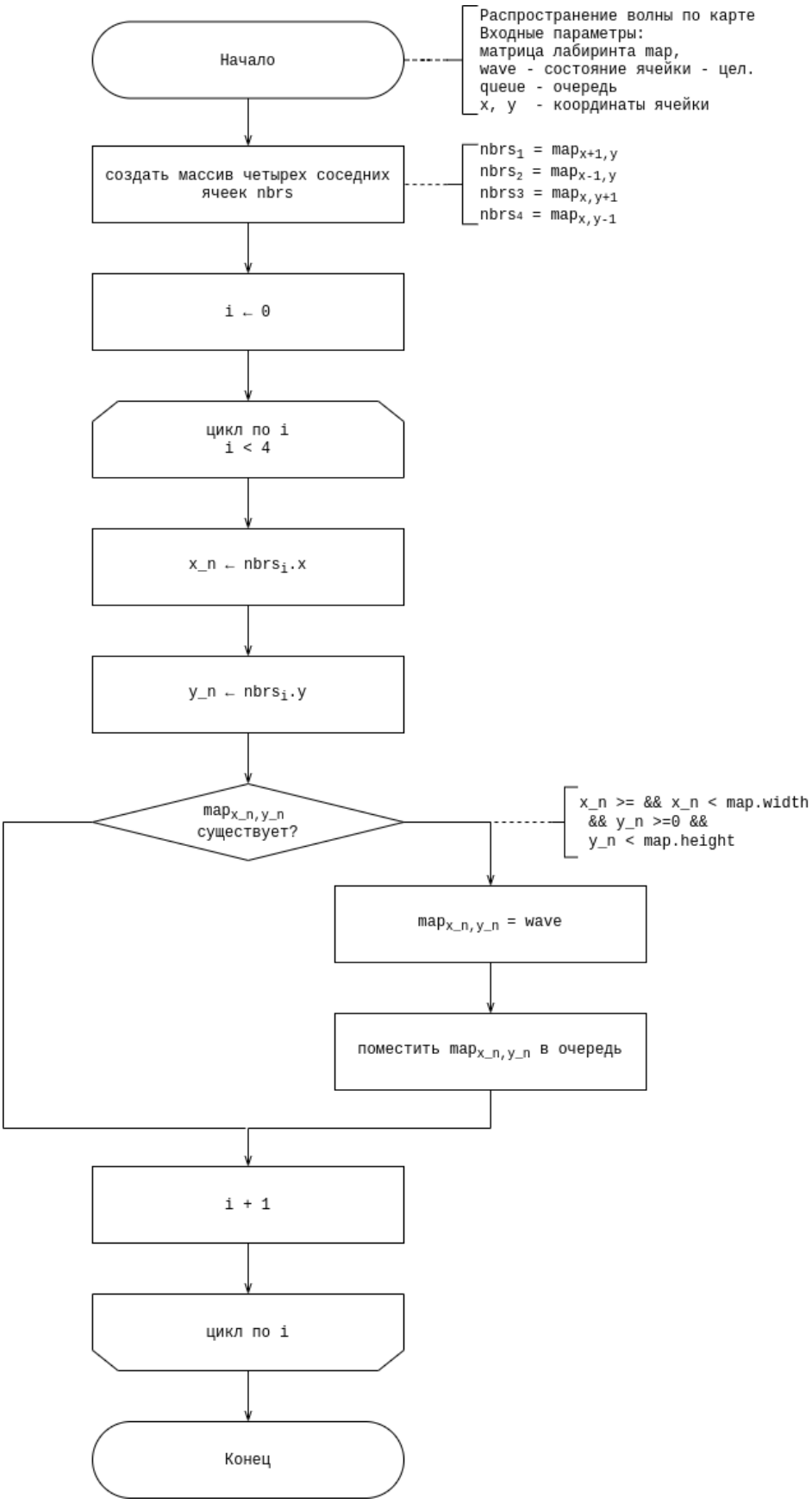


Рисунок 2.3 – Схема процедуры распространения волны

Вывод

Были разработаны схемы последовательного параллельного алгоритма Ли. Получено достаточно теоретической информации для написания программного обеспечения, решающего поставленную задачу.

3. Технологический раздел

3.1 Требования к ПО

Программное обеспечение должно удовлетворять следующим требованиям:

- ПО получает на вход текстовый файл с бинарным лабиринтом размером не превышающим 1000×1000;
- ПО предоставляет пользователю введенный лабиринт, в котором обозначено найденное минимальное расстояние.

3.2 Средства реализации

В качестве языка, на котором будет реализовано программное обеспечение, был выбран язык C++. Выбор языка обусловлен тем, что стандартная библиотека языка имеет существующую реализацию очереди[3] и реализацию работы с нативными потоками[4].

3.3 Листинги кода

Листинг и 3.1 и 3.2 демонстрируют функции – обертки над функцией, выполняющей непосредственно сам поиск в ширину.

Листинг 3.1 – функция - обретка синхронного алгоритма Ли

```
int leeRootSync(Map &map, const coord_t &src, coord_t &dest) {
    CQueue queue;
    queue.push({src, 0});

    bool f = false;
    parallelRoutine(queue, map, dest, f);
    return map.getSlot(dest);
}
```

Листинг 3.2 – функция - обретка параллельного алгоритма Ли

```
int leeRootParallel(Map &map, const coord_t &src, coord_t &dest,
                    int thread_count) {
    CQueue queue;

    queue.push({src, 0});
    std::vector<std::thread> threads;
    bool path_found = false;
    threads.reserve(thread_count);
    for (int i = 0; i < thread_count; i++) {
        threads.emplace_back(
        std::thread(parallelRoutine, std::ref(queue), std::ref(map),
                    dest, std::ref(path_found)));
    }
    for (int i = 0; i < thread_count; i++) {
        threads[i].join();
    }
    return map.getSlot(dest);
}
```

Функция - обертка над распараллеленным алгоритмом включает в себя создание потоков. Сама функция, выполняющая двоичный поиск, представлена в листинге 3.3.

Листинг 3.3 – поиск в ширину

```
void parallelRoutine(CQueue &queue, Map &map, coord_t dest,
                     bool &path_found) {
    while (true) {
        if (path_found) {
            if (queue.size() == 0) {
                return;
            }
            path_found = false;
        }
        auto opt = queue.pop();
        if (!opt)
        continue;

        auto pair = *opt;
        auto coord = pair.first;
        auto val = pair.second + 1;
        if (pair.first == dest) {
            map.setSlot(pair.first, pair.second);
            continue;
        }
        if (!checkCoord(map, coord, val)) {
            if (queue.size() == 0) {
```

```
                path_found = true;
            }
            continue;
        }

        map.setSlot(pair.first, pair.second);

        std::initializer_list<std::pair<coord_t, int>> list = {
            {{coord.first + 1, coord.second}, val},
            {{coord.first - 1, coord.second}, val},
            {{coord.first, coord.second + 1}, val},
            {{coord.first, coord.second - 1}, val}};
        queue.push(list);
    }
}
```

Граф - лабиринт представлен в качестве матрицы, представленной на листингах 3.5 и 3.4.

Листинг 3.4 – Класс матрицы - бинарного лабиринта

```
using coord_t = std::pair<int, int>;
using CQueue = PQueue<std::pair<coord_t, int>>;

class Map {
    public:
    const int WALL = -2;
    const int UNSEEN = -1;

    explicit Map(std::string filename);
    int getSlot(int x, int y);
    int getSlot(std::pair<int, int> c);
    void setSlot(int x, int y, int state);
    void setSlot(std::pair<int, int> c, int state);
    void outputStdput();
    void clear();

    long width() { return width_; };
    long height() { return height_; };

    private:
    long width_;
    long height_;
    std::vector<std::vector<int>> slots_;
    std::mutex mutex_;
};
```

Листинг 3.5 – Реализация класса матрицы - бинарного лабиринта

```
int Map::getSlot(int x, int y) {
    if (x < 0 || y < 0 || x >= width_ || y >= height_)
    return WALL;
    std::lock_guard<std::mutex> guard(mutex_);
    return slots_[x][y];
}
int Map::getSlot(std::pair<int, int> c) {
    return getSlot(c.first, c.second);
}

void Map::setSlot(int x, int y, int state) {
    if (x < 0 || y < 0 || x > width_ || y >= height_)
    throw std::exception();

    std::lock_guard<std::mutex> guard(mutex_);
    slots_[x][y] = state;
}

void Map::setSlot(std::pair<int, int> c, int state) {
    this->setSlot(c.first, c.second, state);
}
```

Реализация очереди, безопасной для распараллеливания, вводится оберткой очереди в класс и инкапсулированием мьютекса, что видно на листинге 3.6.

Листинг 3.6 – Шаблонный класс очереди

```
template<typename T>
class PQueue {
    public:
    PQueue() = default;
    PQueue(const PQueue<T> &) = delete ;
    PQueue& operator=(const PQueue<T> &) = delete ;

    virtual ~PQueue() = default;

    unsigned long size() const {
        std::lock_guard<std::mutex> lock(mutex_);
        return queue_.size();
    }

    std::optional<T> pop() {
        std::lock_guard<std::mutex> lock(mutex_);
        if (queue_.empty()) {
```

```
            return {};
        }
        _i++;
        T tmp;
        if (_i % 2) {
        tmp = queue_.front();
        queue_.pop_front();
    }
    else {
        tmp = queue_.back();
        queue_.pop_back();
    }
    return tmp;
}

void push(const T &item) {
    std::lock_guard<std::mutex> lock(mutex_);

    queue_.push_back(item);
}

template<class P>
void push(const P &list) {
    std::lock_guard<std::mutex> lock(mutex_);
    for (auto &item: list) {
        queue_.push_front(item);
    }
}

private:
    int _i = 0;
    std::list<T> queue_;
    mutable std::mutex mutex_;

    bool empty() const {
        return queue_.empty();
}
};
```

3.4 Тестирование ПО

Таблица 3.1 – Тестовые случаи

№	Матрица	Вход	Выход	Результирующая матрица	Ответ	
					Параллельный	Синхронный
1	1 0 1 1 1 1 0 0 1 1 1 1 0 0 1 1 0 0 1 1 1 1 0 1 1	0 1	4 3	−2 0 −2 −2 −2 −2 1 2 −2 −2 −2 −2 3 4 −2 −2 5 4 −2 −2 −2 −2 5 6 −2	6	6
2	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 1	4 3	1 0 1 2 3 2 1 2 3 4 3 2 3 4 5 4 3 4 5 6 5 4 5 6 7	6	6
3	0	0 0	0 0	0	0	0
4	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	0 1	4 3	−2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2 −2	0	0

3.5 Вывод

Было написано и протестировано программное обеспечение для решения поставленной задачи.

4. Исследовательский раздел

4.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими техническими характеристиками:

- Операционная система Ubuntu 20.04.1 LTS;
- Память 7 GiB
- Процессор Intel(R) Core(TM) i3-8145U CPU @ 2.10GHz [**intel**]
- Процессор имеет 4 ядра.

Во время тестирования устройство было подключено к блоку питания и не нагружено никакими приложениями, кроме встроенных приложений окружения, окружением и системой тестирования.

4.2 Описание системы тестирования

Замеры времени осуществлялись с помощью утилиты $std :: chrono :: system_clock :: now()$ следующим образом:

Листинг 4.1 – Замеры времени

```
1     nowP = std::chrono::system_clock::now();
2     for (int i = 0; i < 10; i++) {
3         map.clear();
4         result = leeRootParallel(map, start, finish, 5);
5     }
6     endP = std::chrono::system_clock::now();
7     nanosecondsP = std::chrono::duration_cast
8                     <std::chrono::nanoseconds>(endP - nowP);
```

4.3 Таблица времени выполнения алгоритмов

Результаты тестирования приведены в таблице 4.1.

Таблица 4.1 – Заметы времени

Размер матрицы	Синх.,ns.	Количество потоков, ns.								
		1	2	4	6	8	10	12	14	16
50	41.5	39.4	57.4	59.3	48.9	42.2	41.6	41.3	42.1	49.5
100	488.4	497.9	629.8	658.4	545.4	457	437.8	400.8	389.7	376.9
200	7747.7	7656.9	7005	5846.1	5652.7	4444.1	5253.9	5285.2	5100.6	5037
300	32772.1	32486.1	20989.8	25498.2	20184.6	17794.7	16904.5	16289.2	15868.4	14504.4
400	94144.1	92434.2	38105	40176.7	52950.5	45672.5	42929.1	40828	64266.4	49693.4

4.4 Графики функций

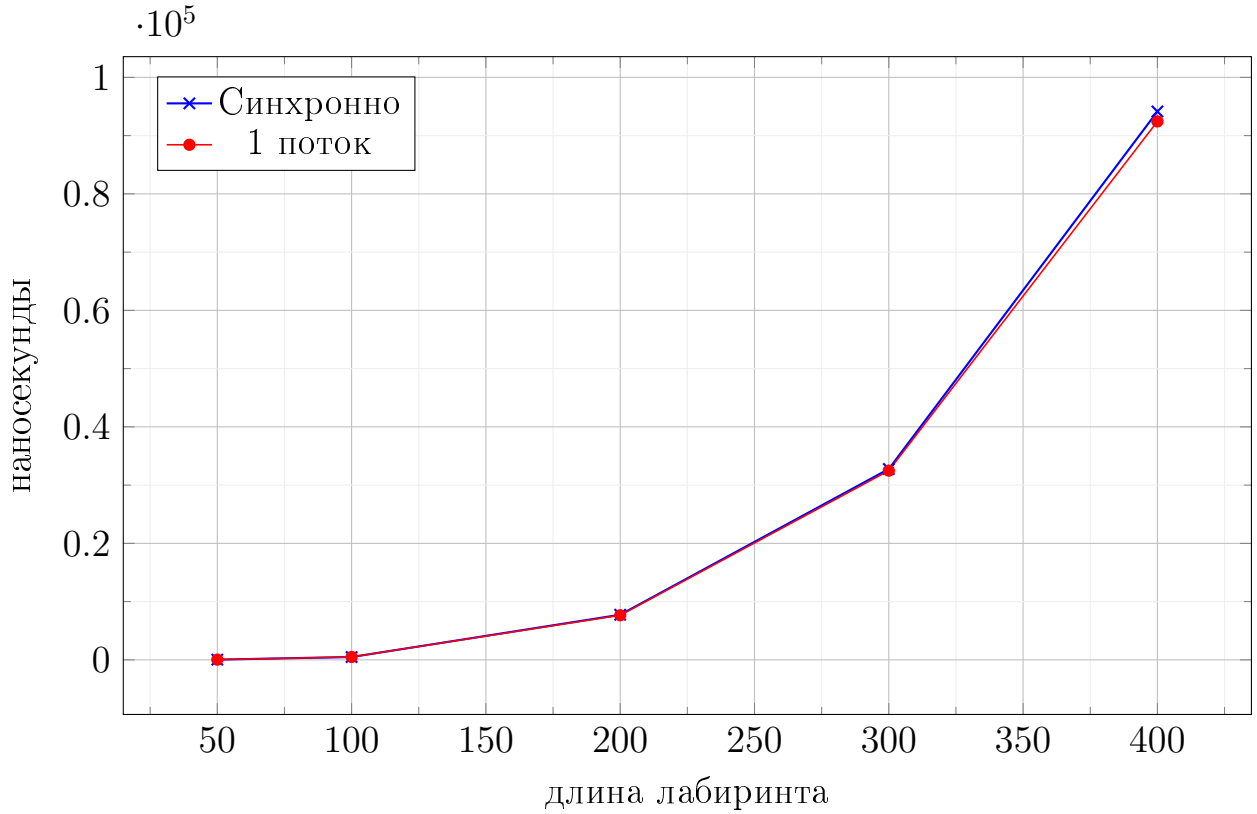


Рисунок 4.1 – Сравнение времени работы синхронной функции и параллельной при одном потоке

При запуске параллельной функции на одном потоке 4.1, синхронная функция работает, очевидно, быстрее – это происходит за счет затрат на обеспечение реентрабельности структур очереди и матрицы а так же на выделение и уничтожение памяти под новые потоки.

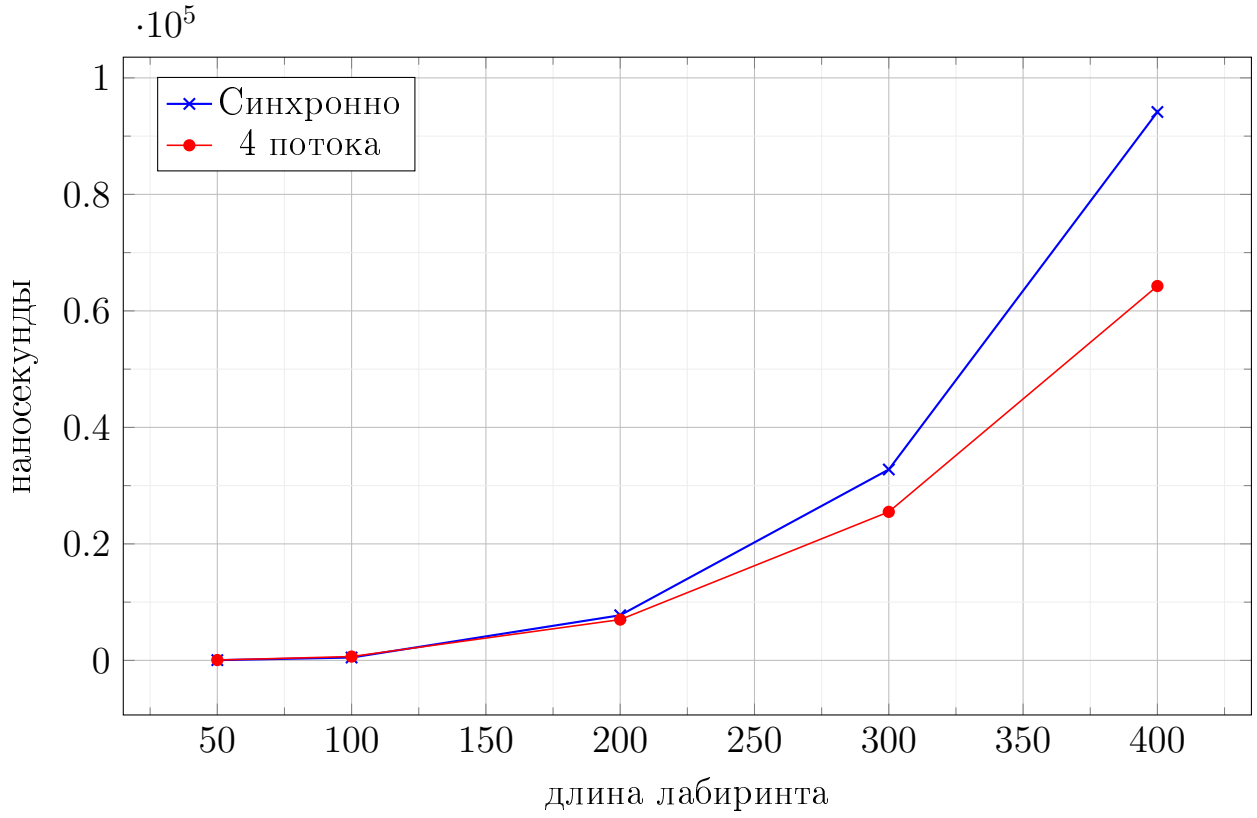


Рисунок 4.2 – Сравнение времени работы синхронной функции и параллельной при одном потоке

При работе алгоритма на количестве потоков, равном количеству логических ядер ЭВМ, на котором проводилось тестирование, с ростом размерности матрицы виден ощутимый прирост(рисунок 4.2) примерно в $\frac{2}{3}$ на самой большой матрице из тестируемых.

4.5 Вывод

Работа с очередью требует использования атомарных операций добавления и удаления вершин в очередь, что на практике может быть источником большого количества накладных расходов. Параллельный алгоритм показывает существенный прирост на таких матрицах, где расходы на эти операции перекрываются трудоемкостью обработки. Соответственно, нативная оптимизация поиска в ширину показывает лучшие результаты на больших объемах данных. Такая оптимизация может получить применение, например, в стратегических играх.

Заключение

Графовые структуры используются повсеместно – от алгоритмов нахождения кратчайшего пути до трассировки печатных плат, поэтому возможности его оптимизации исследуются в неисчислимом множестве научных работ. Алгоритм волновой трассировки на параллельных архитектурах показывает существенный прирост только на матрицах, размерность которых больше, чем 100 элементов. Архитектура устройства, на котором тестировался алгоритм позволяет одному логическому ядру управлять несколькими потоками, поэтому прирост остается ощущаемым на количестве потоков, превышающих количество логических ядер процессора в два раза. Поэтому, ответ на поставленный во введении вопрос неоднозначен – наличие прироста времени с увеличением потоков зависит не только от количества потоков, но и от архитектуры ЭВМ.

Список литературы

[1] C.Y. Lee. “An algorithm for path connections andits applirations”. в: *IRE Trans. Electronac Computers* (1961).

[2] Левитин А. В. “Алгоритмы. Введение в разработку и анализ”. в: М.: «Вильямс», 2006. гл. 5: Метод уменьшения размера задачи: Поиск в ширину.

[3] *std::queue*. URL: `https://en.cppreference.com/w/cpp/container/queue` (дата обр. 09.10.2021).

[4] *std::thread*. URL: `https://en.cppreference.com/w/cpp/thread/thread` (дата обр. 09.10.2021).