МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Южно-Уральский государственный университет
(национальный исследовательский университет)»**
Высшая школа электроники и компьютерных наук
Кафедра системного программирования

РАБОТА ПРОВЕРЕНА	ДОПУСТИТЬ К ЗАЩИТЕ
Рецензент к. ф.-м. н, доцент кафедры ВМиИТ ЧелГУ А. Ю. Маковецкий	Заведующий кафедрой, д.ф.-м.н., профессор __________ Л.Б. Соколинский “___”___________ 2019 г.

РАЗРАБОТКА И РЕАЛИЗАЦИЯ ПАРАЛЛЕЛЬНОГО АЛГОРИТМА ДЛЯ ВЫЧИСЛЕНИЯ ЛИНЕЙНОЙ ВАРИАЦИИ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
ЮУрГУ – 02.03.02.2019.115-122.ВКР

Научный руководитель,
к.ф.-м.н., доцент кафедры СП
Т. Ю. Маковецкая

Автор работы,
студент группы КЭ-401
А. А. Тюрин

Ученый секретарь
(нормоконтролер)
_____________ О.Н. Иванова
“___”___________ 2019 г.

Челябинск-2019

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Южно-Уральский государственный университет
(национальный исследовательский университет)»**
Высшая школа электроники и компьютерных наук
Кафедра системного программирования

УТВЕРЖДАЮ
Зав. кафедрой СП
__________ Л.Б. Соколинский
09.02.2019

ЗАДАНИЕ
на выполнение выпускной квалификационной работы бакалавра
студенту группы КЭ-401
Тюрину Артёму Андреевичу,
обучающемуся по направлению
02.03.02 «Фундаментальная информатика и информационные технологии»

1. **Тема работы** (утверждена приказом ректора от «25» апреля 2019 № 899)
Разработка и реализация параллельного алгоритма для вычисления линейной вариации.

2. **Срок сдачи студентом законченной работы:** 05.06.2019.

3. **Исходные данные к работе**

3.1. Воеводин В. Параллельные вычисления. – СПб.: БХВ–Петербург, 2002. – 608 с.
3.2. Alexeev F., Alexeev M., Makovetskii A. Linear Variation and Optimization of Algorithms for Connected Components Labeling in Binary Images. – Moscow: Moscow Institute of Physics and Technology State University, 2014. – 11 p.
3.3. Антонов А. Параллельное программирование с использованием технологии OpenMP. – М.: Издательство московского университета, 2009. – 77 с.

4. Перечень подлежащих разработке вопросов

4.1. Осуществить обзор литературы и существующих решений.
4.2. Реализовать последовательную версию алгоритма для вычисления линейной вариации.
4.3. Реализовать параллельную версию алгоритма для вычисления линейной вариации.
4.4. Провести численные эксперименты и оценку результатов

Дата выдачи задания: 08.02.2019.

Научный руководитель
к.ф.-м.н., доцент кафедры СП Т. Ю. Маковецкая
Задание принял к исполнению А. А. Тюрин

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ....5

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ....7

1.1. Предметная область....7

1.2. Обзор алгоритмов для решения задачи подсчета компонент связности9

1.3. Технологии реализации параллельности....11

2. ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ....14

2.1. Проектирование последовательного алгоритма....14

2.2. Реализация последовательного алгоритма....18

2.3. Проектирование параллельного алгоритма....20

2.2. Реализация параллельного алгоритма....21

3. ВЫЧИСЛИТЕЛЬНЫЕ ЭКСПЕРИМЕНТЫ....24

3.1. Исходные данные для тестирования....24

3.2. Сравнение работы алгоритмов....25

ЗАКЛЮЧЕНИЕ....31

ЛИТЕРАТУРА....32

ВВЕДЕНИЕ

АКТУАЛЬНОСТЬ РАБОТЫ

За последние десятки лет компьютерное зрение развилось из узкоспециализированного направления исследований, сфокусированных на решении прикладных задач, в независимый раздел в области искусственного интеллекта [2]. На сегодняшний день компьютерное зрение сочетает в себе большое количество различных подходов к машинному обучению, линейные алгоритмы и нейросетевые технологии.

Стремительное развитие данной отрасли связано с технологическим прогрессом и широким распространением цифровых камер, способных снимать изображение в достаточном для большинства систем компьютерного зрения качестве [8].

Помимо этого, на развитие систем компьютерного зрения также повлияло повсеместное повышение мощностей персональных компьютеров, сделав более доступными для конечных пользователей технически требовательные программные продукты, в том числе, использующие параллельную реализацию [12].

В современных реалиях, когда количество ядер в процессорах персональных компьютеров, все большее число программных продуктов ориентируется на распараллеливание с целью оптимизации времени работы программ, а параллельные алгоритмы показывают себя лучше, чем более эффективные последовательные аналоги.

За последние годы, в связи с применимостью в различных прикладных областях, в области компьютерного зрения становится более востребованной задача подсчета количества схожих предметов на изображении, одним из решений которой является подсчет компонент связности в графе, представляющем изображение, или так называемая задача вычисления линейной вариации.

В данной работе предлагается параллельный метод решения задачи линейной вариации, не имеющий узкой специализации и применимый для решения широкого круга прикладных задач.

ЦЕЛЬ И ЗАДАЧИ РАБОТЫ

Целью данной работы является разработка и реализация параллельного алгоритма для вычисления линейной вариации.

Для достижения поставленной цели были вынесены следующие задачи:

1) осуществить обзор литературы и существующих решений;
2) реализовать последовательную версию алгоритма для вычисления линейной вариации;
3) реализовать параллельную версию алгоритма для вычисления линейной вариации;
4) провести численные эксперименты и оценку результатов.

СТРУКТУРА И ОБЪЕМ РАБОТЫ

Данная работа состоит из введения, четырех глав, заключения и библиографии. Объем работы составляет 33 страницы, объем библиографии – 20 источников.

СОДЕРЖАНИЕ РАБОТЫ

В первой главе рассматривается предметная область, в рамках которой выполняется данная работа. Помимо этого, глава содержит обзор существующих алгоритмов, предназначенных для достижения целей, схожих с целью данной работы.

Во второй главе приведены теоретические основы данной работы, а также описаны технологии, использованные для реализации параллельных вычислений.

Третья глава содержит полное описание проделанной работы по реализации параллельного алгоритма для вычисления линейной вариации.

Четвертая глава представляет собой описание проведенных вычислительных экспериментов, а также их результаты и оценку.

В заключении подытожены результаты работы и приводятся возможные направления ее дальнейшего развития.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Предметная область

На сегодняшний день компьютерное зрение решает широкий круг прикладных задач, однако, большинство из них так или иначе относятся к одной из следующих групп: поиск, распознавание или сегментирование изображений.

В рамках данной работы мы реализуем программную систему, предназначенную для вычисления линейной вариации, основанную на параллельном алгоритме, и позволяющую в условиях, ограниченных производительностью персонального компьютера, осуществлять работу с изображениями различных размеров.

Решение задачи вычисления линейной вариации для заданного изображения является основой для работы в различных направлениях компьютерного зрения, в том числе, восстановлении, сегментации, поиске на изображениях [5].

Математическая постановка задачи

Вариация множества – число, характеризующее k-мерную протяженность множества в n-мерном евклидовом пространстве. Нулевая вариация $v_0(E)$ замкнутого ограниченного множества E есть число компонент этого множества [4].

Линейная вариация – топологическая характеристика функции двух переменных [14]. Для множества E она определяется как

$$v_1(E) = c \int_0^{2\pi} v_0(E, L_\alpha) d\alpha,$$

где L_α – прямая $x \cos \alpha + y \sin \alpha = 0$, c – константа [4]. Тогда на двумерном изображении линейная вариация будет являться суммарным числом компонент, составленных из соседних элементов с совпадающими значениями, для различных значений градаций яркости.

В рамках представления изображения в качестве графа такие компоненты будут являться компонентами связности, и, соответственно, задача вычисления линейной вариации в таком случае сводится к задаче подсчета,

или выделения компонент связности для различных графов, составленных для на основании уровней яркости.

Компонентой связности графа G называется его подграф, не являющийся собственным подграфом никакого другого связного подграфа графа G [16].

Программное представление

В разрабатываемой программной системе на вход планируется подавать изображения в формате PNG либо JPEG (JPG). На сегодняшний день форматы PNG и JPEG являются наиболее распространенными для представления растровых изображений, помимо этого, большинство других распространенных форматов поддерживают конвертацию в JPG или PNG [9].

Полученное после ввода изображения конвертируется в цветовую палитру оттенков серого так, что каждый пиксель будет представлен в программе числом от 0 до 255.

Данную матрицу, представляющую изображение, можно представить в виде взвешенного неориентированного графа с «раскрашенными» вершинами, расстояние между двумя соседними которыми будет нами принято равным 1.

В рамках данной работы смежными вершинами, образующими компоненту связности при условии одинаковости окраски, будут считаться вершины, расположенные на расстоянии 1 по вертикали и горизонтали и $\sqrt{2}$ по диагонали.

Таким образом, для вычисления линейной вариации необходимо представить изображение как граф, и для каждого его подграфа, представляющего определенный цвет, решить задачу выделения компонент связности.

Впоследствии полученные сегментацией данные могут быть использованы для машинного анализа исходного изображения, развития алгоритма для решения различных задач компьютерного зрения.

Далее в данной главе будут рассмотрены различные подходы к решению задачи выделения компонент связности и реализации параллельных программ.

1.2. Обзор алгоритмов для решения задачи подсчета компонент связности

Для решения задачи подсчета компонент связности, к которой частично была сведена задача вычисления линейной вариации, на сегодняшний день применяется широкий спектр алгоритмов. В данном разделе будут рассмотрены наиболее подходящие для данной работы алгоритмы, а также будет осуществлен выбор алгоритма для его дальнейшей реализации.

Поиск в глубину и поиск в ширину

Поиск в глубину, или Depth-First Search (DFS) является одним из основных методов обхода графа, поиска любого пути в графе, поиска лексикографически первого пути в графе. Помимо этого, поиск в глубину также применим для решения задачи поиска компонент связности в графе.

Алгоритм поиска в глубину будет звучать следующим образом.

Пусть зафиксирована начальная вершина v_0. Тогда

1) выберем смежную с ней вершину v_1;

2) для вершины v_1 выбираем смежную с ней вершину из числа еще невыбранных вершин;

3) если мы уже выбрали вершины $v_0,v_1,...,v_k$, то следующая вершина выбирается смежной с вершиной v_k из числа невыбранных;

4) если для вершины v_k такой вершины не нашлось, то возвращаемся к вершине v_{k-1}, и для нее ищем смежную среди невыбранных.

При необходимости возвращаемся назад. Так будут перебраны все вершины графа и поиск завершится [13].

Для решения задачи поиска компонент связности достаточно базовой реализации поиска в глубину, основанной на рекурсивных вызовах, с отметкой пройденных вершин.

Преимуществами данного алгоритма являются простота реализации и работы с графами малого размера. Однако, необходимость использования рекурсивных вызовов для каждой из вершин графа делает его оптимальное распараллеливание близким к невозможному, а время работы для крупных изображений является существенно ухудшается [9]

Особенности реализации, преимущества и недостатки алгоритма поиска в ширину (Breadth-first search или BFS) аналогичны таковым у алгоритма DFS.

Алгоритм, основанный на реализации системы непересекающихся множеств

Система непересекающихся множеств (Disjoint-set-union или DSU) является структурой данных, которая представляет собой набор элементов, изначально находящихся в собственных отдельных множествах. За одну операцию возможно объединение каких-либо двух множеств, либо поиск множества, в котором находится указанный элемент, либо добавление нового элемента, находящегося в его собственном множестве [1].

Множества элементов хранятся в виде деревьев, одно дерево соответствует одному множеству. Корень дерева также является представителем множества, на которого ссылаются остальные элементы.

Система непересекающихся множеств необходима нам для решения задачи поиска компонент связности на изображении. В таком случае на ее основе реализуется алгоритм объединения соседних компонент с одинаковым значением в одну общую.

Для решения поставленной задачи необходимо перебирать все элементы матрицы, которые являются целыми числами, представляющими собой оттенок серого в 8-битной градации, и для каждого значения матрицы серого проверять четырех его соседей (для $a_{i,j}$ это будут $a_{i+1,j}$, $a_{i-1,j}$, $a_{i,j+1}$ и $a_{i,j-1}$ соответственно), по их значению. Если значение соседа совпадает со значением проверяемой вершины и не принадлежит текущему

множеству, множества, которым принадлежат эти вершины объединяются в одно посредством замены представителя одного из множеств.

Получившиеся по завершению полного обхода деревья будут являться искомыми компонентами связности.

В результате сравнения данного алгоритма с алгоритмами поиска в глубину и поиска в ширину был сделан выбор в пользу реализации собственного алгоритма поиска и объединения, основанного на DSU, в связи с более оптимальной работой с изображениями большого размера, а также возможностью полноценной реализации параллельности, и более эффективным временем работы на изображениях большого размера.

1.3. Технологии реализации параллельности

Основной целью параллельных вычислений является ускорение решения вычислительных задач [15].

В рамках работы параллельной программы осуществляется управление работой множества процессов, организуется обмен данных между процессами, задачи разбиваются на подзадачи и выполняются параллельно.

В данном разделе рассматриваются наиболее распространенные технологии реализации параллельных вычислений, а также осуществляется выбор наиболее походящего для данной работы варианта.

MPI

В вычислительных системах с распределенной памятью процессоры работают независимо друг от друга. С целью организации параллельных вычислений в таких условиях существует необходимость иметь возможность распределять вычислительную нагрузку и организовать информационное взаимодействие между процессорами [18]. Решение данных вопросов предлагается интерфейсом передачи данных MPI.

Под параллельной программой в рамках MPI понимается множество одновременно выполняемых процессов. Процессы могут выполняться как на разных, так и на одном процессоре. На одном процессоре могут

располагаться также несколько процессов (в этом случае их исполнение осуществляется в режиме разделения времени).

Основу MPI составляют операции передачи сообщений. Среди предусмотренных в составе MPI функций различаются парные операции между двумя процессами и коллективные коммуникационные действия для одновременного взаимодействия нескольких процессов.

CUDA

CUDA – это архитектура параллельных вычислений, разработанная компанией NVIDIA, позволяющая существенно увеличить вычислительную производительность благодаря использованию графических процессоров для осуществления вычислений. [19]

Основная особенность архитектуры CUDA заключается в предоставлении разработчику возможности проведения вычислений и управления памяти на графическом ускорителе.

CUDA позволяет программистам реализовывать на специальном диалекте языка C алгоритмы, которые используются в графических процессорах NVIDIA, и включать специальные функции в текст программы на C.

OpenMP

OpenMP – стандарт распараллеливания программ, реализованный на языках C, C++, Fortran [11]. OpenMP является механизмом написания параллельных программ для систем с общей памятью и состоит из набора директив компилятора и библиотечных функций.

Модель работы OpenMP – fork-join, метод, заключающийся в разбиении каждой задачи на множество более мелких синхронизированных задач, обрабатываемых параллельно [3].

Программирование на OpenMP происходит путем вставки директив компилятора в исходный код программы. Данные директивы интерпретируются компилятором с целью последующей вставки вызовов из библиотек, приводящих к параллельному выполнению указанных участков кода. При выборе мест распараллеливания стоит помнить о равномерном

распределении вычислительной нагрузки, так как оптимизация данного параметра напрямую влияет на получаемое ускорение от распараллеливания.

Для контроля ошибок доступа к общим ресурсам (так называемые ошибки соревнования или race conditions) в OpenMP применяются простейшие механизмы синхронизации, такие, как критические секции, барьеры, атомарные операции и блокировки.

Выбор технологии распараллеливания

Выбор технологии распараллеливания был осуществлен на основе анализа существующих технологий, с учетом условий применения данного программного продукта конечным пользователем.

Необходимо создать доступный программный продукт, не зависящий от технического оснащения рабочего места конечного пользователя. Целевой пользователь данной системы оперирует персональным компьютером с ограниченными техническими характеристиками, в связи с этим, отсутствует необходимость использования MPI, оптимального для системы с большим количеством вычислителей. Также, не является фиксированным и наличие у конечного пользователя графического процессора производителя NVIDIA. Данное ограничение отнимает возможность реализации системы с использованием технологии CUDA.

Высокая вариативность возможностей использования реализуемой системы диктует требование к слабой зависимости от технических характеристик рабочего места конечного пользователя. Таким образом, для реализации параллельности был выбран стандарт OpenMP.

Выводы по первой главе

В рамках первой главы была определена предметная область, дана математическая постановка задачи. Помимо этого, были рассмотрены существующие алгоритмы решения данной задачи, осуществлен выбор технологии для реализации параллельной системы.

2. ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ

2.1. Проектирование последовательного алгоритма

Предобработка данных для алгоритма устроена следующим образом. На вход подается изображение произвольного размера в формате PNG либо JPG.

Цветовая палитра входного изображения конвертируется в палитру оттенков серого, где каждый из пикселей представлен целочисленным значением от 0 до 255, характеризующим его цвет.

Из полученного изображения необходимо получить 8 уровней яркости. Сделать это можно различными способами – упрощением цветов, осветлением либо затемнением изображения.

При упрощении цветов мы целочисленно делим значение каждого элемента матрицы на 2^n, где n – от 1 до 7. При таком подходе на первом уровне яркости у нас будет 128 оттенков серого, а на последнем – 2.

При осветлении либо затемнении изображения мы соответственно прибавляем, либо вычитаем $\frac{256}{16} = 32$ от каждого значения в матрице, однако не более 255 и не менее 0. Таким образом, на последнем уровне яркости изображение будет полностью белым либо черным соответственно.

Следующий шаг в предобработке изображения – генерация структуры DSU для каждого элемента массива, представляющего собой цвет соответствующего пикселя на изображении. Данный шаг завершает часть, связанную с предустановками перед началом работы алгоритма.

Следующий этап – проход по исходному изображению, а также изображениям, представляющим уровни яркости, с целью определения смежных вершин одного цвета и добавления их в общие компоненты связности.

Рассмотрим работу алгоритма проверки принадлежности и условия объединения на примере графа 3 на 3, визуализированного на рис. 1.

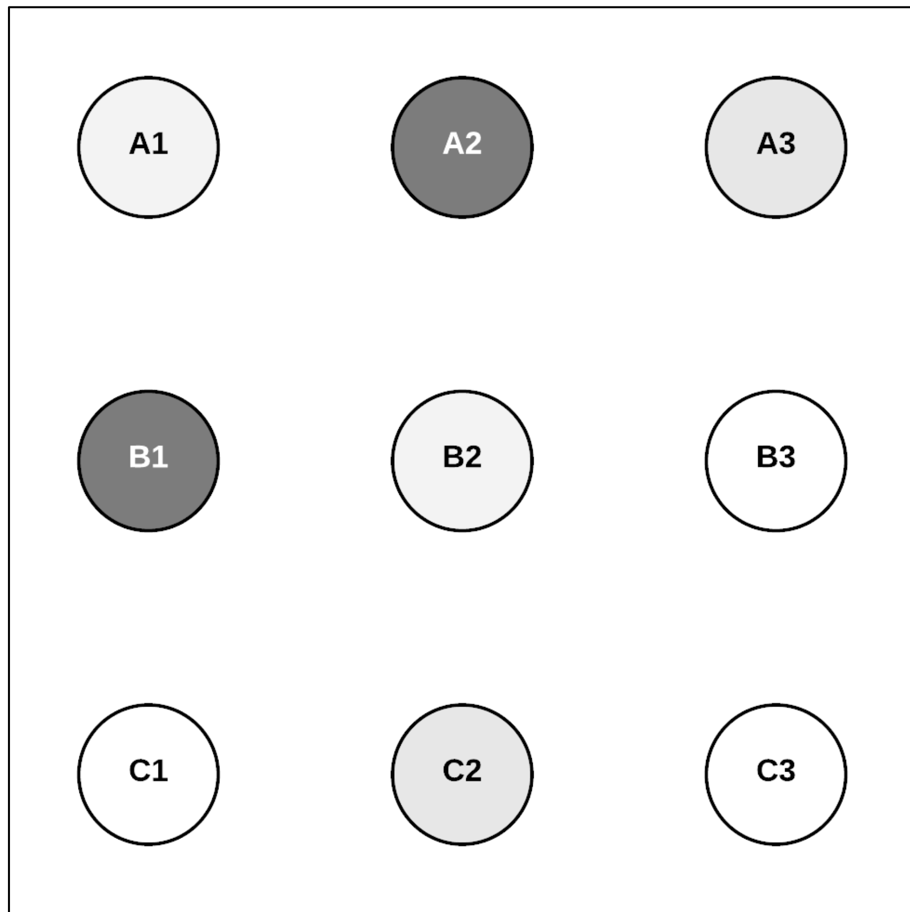

Рис. 1. Теоретический граф с вершинами, раскрашенными в разные цвета

В данном примере вершины одного цвета должны по итогу работы алгоритма принадлежать к одной компоненте связности, при условии, что они являются соседними.

Так как проход по массиву совершается слева направо и сверху вниз, то для $a_{i,j}$ нам достаточно проверить $a_{i+1,j}$, $a_{i,j+1}$, $a_{i+1,j+1}$, $a_{i-1,j+1}$ (если данная вершина существует для $a_{i,j}$). Отношение остальных вершин на минимальном расстоянии к данной при таком подходе уже будет рассмотрено ранее.

Таким образом, после обработки данным алгоритмом представленного ранее теоретического графа, будет найдено 4 компоненты связности, одна для цвета, соответствующего цвету вершины A1, один для вершины A2, и два для B3. Смежные вершины одного цвета объединяются в компоненты связности при помощи операции объединения DSU.

Результат для рассмотренной теоретической ситуации приведен на рис. 2.

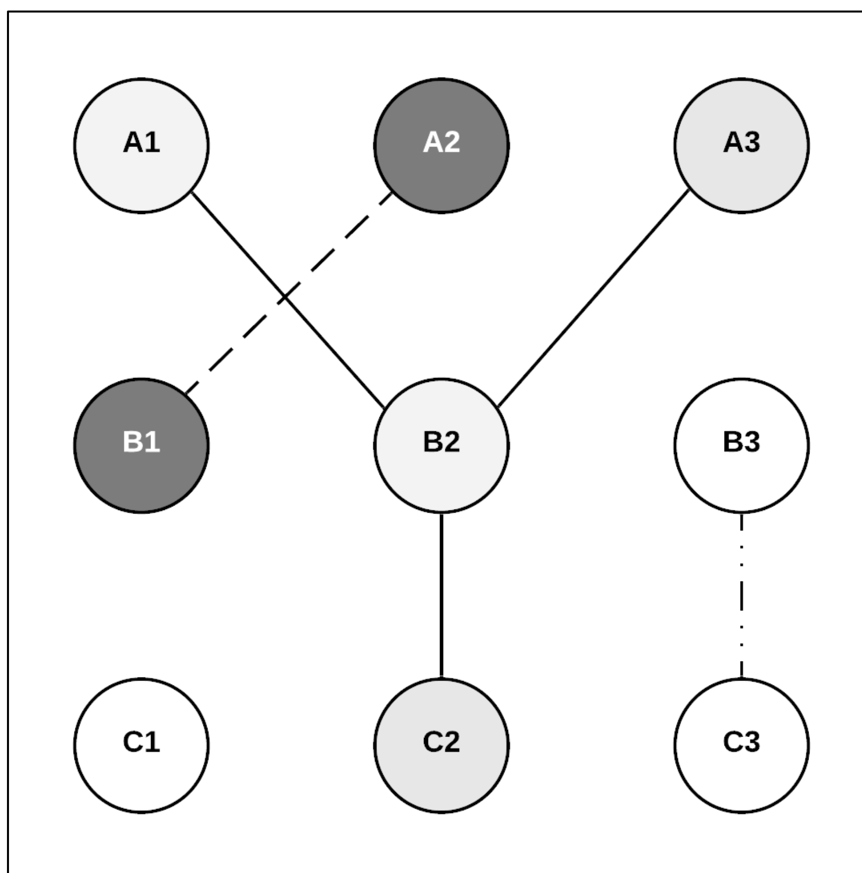

Рис. 2. Компоненты связности на приведенном ранее теоретическом графе

Блок-схема данного алгоритма приведена на рис. 3.

Данная блок-схема отражает реализацию поиска и объединения компонент связности в системе непересекающихся множеств. Для матрицы размера n на m она описывает $n*m$ операций, осуществляемых алгоритмом, с проверкой цветов соседних вершин в каждой из них.

Таким образом, возможно рассчитать предполагаемую сложность приведенного алгоритма, которая должна составить $k * O(n \times m) + c$, где n, m – размеры исходного изображения, k и c – некоторые константы, напрямую зависящие от реализации алгоритма. Сложность разработанного алгоритма больше сложности аналогов, однако, возможность его параллельной реализации позволяет предполагать о снижении сложности части алгоритма в p раз, где p – число вычислителей компьютера пользователя.

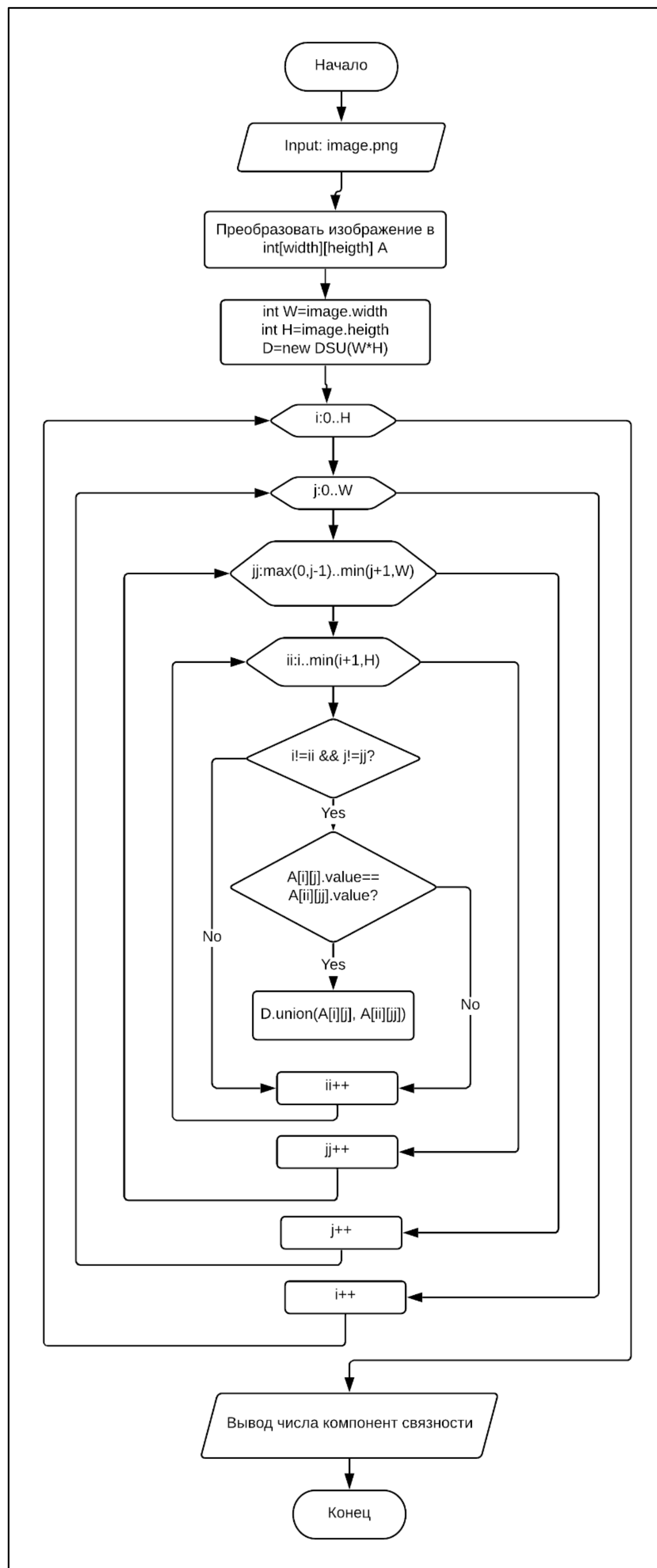

Рис. 3. Блок-схема реализованного алгоритма

2.2. Реализация последовательного алгоритма

Основой для работы последовательной и параллельной версий алгоритма является реализация структуры данных DSU, на основании которой впоследствии производится поиск и разметка компонент связности.

Для поиска компонент связности в DSU требуются все стандартные операции, реализуемые с этой структурой данных, а именно создание множества, поиск представителя в множестве, объединение множеств.

Листинг реализации системы непересекающихся множеств приведен на рис. 4.

```
class DSU {
private:
  vector<int> parent;
  vector <int> rank;
  int size = 0;
public:
  void make_set(int v) {
    if (parent.size() < v) {
      for (int i = size; i < v; i++) {
        parent.push_back(0);
        rank.push_back(0);
      }
      size = v;
    }
    parent[v - 1] = v;
    rank[v - 1] = 0;
  }

  int find_set(int v) {
    if (v == parent[v - 1])
      return v;
    return parent[v - 1] = find_set(parent[v - 1]);
  }

  void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
      if (rank[a - 1] < rank[b - 1])
        swap(a, b);
      parent[b - 1] = a;
      if (rank[a - 1] == rank[b - 1])
        ++rank[a - 1];
    }
  }
};
```

Рис. 4. Листинг реализации DSU

Для работы с DSU необходима предобработка входного изображения. При помощи библиотеки OpenCV [17] изображение конвертируется в палитру оттенков серого, после чего происходит создание 8 новых изображений, представляющих собой различные уровни яркости исходного изображения.

Для выделения уровней яркости используется описанные ранее подход, при котором каждое значение исходной матрицы целочисленно делится на 2^n, где n – номер текущего уровня яркости.

По выполнению предобработки преобразованные изображения последовательно подаются на вход алгоритму поиска, составляющему и подсчитывающему общее число компонент связности для каждого уровня яркости, описанному ранее.

Листинг последовательной реализации алгоритма приведен на рис. 5.

```
void DSU_serial(Mat a, char img_type, int num){//int a[heigth][width]) {
   DSU d;
   for (int i = 0; i < a.rows; ++i)
      for (int j = 0; j < a.cols; ++j)
         d.make_set(get_num(i, j, a.cols));
   for (int i = 0; i < a.rows-1; ++i) {
      for (int j = 0; j < a.cols-1; ++j) {
         for (int jj = max(0, j - 1); jj <= min(a.cols, j + 1); ++jj) {
            for (int ii = i; ii <= min(a.rows, i + 1); ++ii) {
               if (ii != i || jj != j) {
                  if ((int)a.at<uchar>(i, j) == (int)a.at<uchar>(ii, jj)){//(a[i][j]
== a[ii][jj]) {
                     d.union_sets(get_num(i, j, a.cols), get_num(ii, jj, a.cols));
                  }
               }
            }
         }
      }
   }

   set<int> CC;
   for (int i = 0; i < a.rows; ++i)
      for (int j = 0; j < a.cols; ++j)
         CC.insert(d.find_set(get_num(i, j, a.cols)));
```

Рис. 5. Листинг последовательной реализации алгоритма

Расчетная сложность приведенного алгоритма составляет $8 * O(n \times m) + 4$, где n, m – размеры исходного изображения.

2.3. Проектирование параллельного алгоритма

Для создания параллельного алгоритма было решено использовать полноценную последовательную реализацию, частично выполняемую дополнительными нитями.

Одним из основных вопросов, стоящих при проектировании параллельной версии алгоритма, является вопрос выбора части, которая подвергнется разделению на несколько нитей.

В связи с тем, что последовательный алгоритм осуществляет построчную обработку матрицы, представляющую собой изображение, было решено разделить данную матрицу на некоторое количество блоков строк, и затем поручить обработку этих блоков параллельно запущенным версиям алгоритма.

Таким образом появляется возможность поручить создаваемым нитям выполнение задач, имеющих близкую к одинаковой нагрузку, и могут выполняться за схожее время. Для этого необходимо разделить высоту исходной матрицы h на количество создаваемых потоков *p,* и поручить каждому потоку обработку $\frac{h}{p}$ числа строк матрицы.

В данном алгоритме параллельно выполняющиеся нити не имеют возможности обратиться к общему ресурсу, и, соответственно, отсутствует необходимость в использовании простейших средств синхронизации. Однако, существует возможность неверного срабатывания алгоритма в случае, если две нити одновременно попробуют объединить компоненту, находящуюся одновременно в двух блоках, обрабатываемых различными нитями.

Во избежание данной ситуации каждой нити, кроме той, которая должна обработать последние строки матрицы, дается задание на обработку $\frac{h}{p} - 1$ строк. Тогда при обработке каждая нить будет проверять последнюю строку своего блока строк из соседних вершин, однако не начнет проверку из вершин самой строки. По завершению работы всех созданных потоков

главный поток запустит алгоритм для каждой из таких строк, что приведет к объединению результатов работы нитей.

Предполагаемая сложность приведенного алгоритма должна составить $k * O(n \times m) + t * \frac{O(n \times m)}{p} c$, где n, m – размеры исходного изображения, p – количество создаваемых потоков.

2.2. Реализация параллельного алгоритма

Для создания параллельной реализации алгоритма использовался открытый стандарт OpenMP [7].

В соответствии с проектированием, при помощи директивы *#pragma omp parallel* OpenMP, измененный последовательный алгоритм запускается для параллельной работы на некотором числе потоков, разделение ресурсов между которыми стабильно, и не существует возможности нарушения их целостности. Выбор директивы *parallel* вместо *parallel for* продиктован необходимостью проверить, не является ли данная строка последней в блоке. При такой реализации использование директивы *parallel for* невозможно.

Количество создаваемых потоков определяется функцией OpenMP *omp_get_max_threads()*, возвращающей расчетное оптимальное количество потоков для создания, зависящее от числа физических и виртуальных вычислителей на конкретной ЭВМ.

Выделение количества строк потокам не является равномерным для любого случая, так как в ситуациях, когда существует остаток от деления $\frac{h}{p}$, его необходимо отдать на обработку одной из нитей.

По завершению работы всех нитей, а также объединения результатов, алгоритм переходит к обработке матрицы, отвечающей за следующий уровень яркости. В связи с тем, что нам необходимо обработать каждый элемент в каждой матрице, не обязательно запускать обработку каждого изображения параллельно. Реализация данной системы ограничена условиями

обычного персонального компьютера, где оптимальное количество создаваемых потоков обычно не будет превышать количества ядер больше, чем в два раза.

Листинг параллельной реализации алгоритма приведен на рис. 6.

```
void DSU_parallel(Mat a, char img_type, int num){
  DSU d;
  for (int i = 0; i < a.rows; ++i)
    for (int j = 0; j < a.cols; ++j)
      d.make_set(get_num(i, j, a.cols));
  int thread_num = omp_get_max_threads();
  int slice = a.rows / thread_num;
  omp_set_num_threads(thread_num);

#pragma omp parallel
  {
  int current_thread = omp_get_thread_num();
  if (current_thread == thread_num - 1) {
    for (int i = current_thread * slice; i < a.rows-1; ++i) {
      for (int j = 0; j < a.cols-1; ++j) {
          for (int jj = max(0, j - 1); jj <= min(a.cols, j + 1); ++jj) {
           for (int ii = i; ii <= min(a.rows, i + 1); ++ii) {
            if (ii != i || jj != j) {
                                d.union_sets(get_num(i,         j),
get_num(ii, jj));
    }}}}}}}
    else {
      for (int i = current_thread * slice; i < (current_thread + 1) *
slice - 1; ++i) {
        for (int j = 0; j < a.cols-1; ++j) {
          for (int jj = max(0, j - 1); jj <= min(a.cols, j + 1); ++jj) {
            for (int ii = i; ii <= min(a.rows, i + 1); ++ii) {
              if (ii != i || jj != j) {
                if ((int)a.at<uchar>(i, j) == (int)a.at<uchar>(ii, jj))
{
                  d.union_sets(get_num(i, j, a.cols), get_num(ii, jj,
a.cols));
                }}}}}}}}
for (int i = slice - 1; i < a.rows-1; i += slice) {
    for (int j = 0; j < a.cols-1; ++j) {
      for (int jj = max(0, j - 1); jj <= min(a.cols, j + 1); ++jj) {
        for (int ii = i; ii <= min(a.rows, i + 1); ++ii) {
          if (ii != i || jj != j) {
            /*if (a[i][j] == a[ii][jj]) {
              d.union_sets(get_num(i, j), get_num(ii, jj));
            }*/
            if ((int)a.at<uchar>(i, j) == (int)a.at<uchar>(ii, jj)) {
              d.union_sets(get_num(i,  j,  a.cols),  get_num(ii,  jj,
a.cols));
            }}}}}}}
```

Рис. 6. Листинг параллельной реализации алгоритма

Вычислительная сложность представленного алгоритма составляет $3 * O(n \times m) + \frac{5*O(n \times m)}{p} + 2$, где n, m – размеры исходного изображения, p – количество вычислителей у пользователя.

Выводы по третьей главе

В данной главе были спроектированы и реализованы предложенные последовательный и параллельный алгоритмы для вычисления линейной вариации.

3. ВЫЧИСЛИТЕЛЬНЫЕ ЭКСПЕРИМЕНТЫ

Вычислительный эксперимент — метод изучения устройств или физических процессов с помощью математического моделирования. Он предполагает, что вслед за построением математической модели проводится ее численное исследование, позволяющее изучить поведение исследуемого объекта в различных условиях или в различных модификациях [20].

Численное исследование модели дает возможность определять разнообразные характеристики процессов, оптимизировать конструкции или режимы функционирования проектируемых устройств. Более того, случается, что в ходе вычислительного эксперимента исследователь неожиданно открывает новые процессы и свойства, о которых ему ранее ничего не было известно.

Ускорением параллельного алгоритма называют отношение времени выполнения лучшего последовательного алгоритмам к времени выполнения параллельного алгоритма [18]:

$$S = \frac{T_1}{T_p} \text{ [12].}$$

Помимо этого, для оценки масштабируемости, то есть возможности обеспечения увеличения ускорения при росте числа процессоров (при сохранении постоянного уровня эффективности использования процессоров) параллельного алгоритма используется следующая формула:

$$E = \frac{S}{P}.$$

3.1. Исходные данные для тестирования

Вычислительные эксперименты были проведены на устройствах с различной конфигурацией с целью определения эффективности работы разработанного алгоритма в различных условиях. В табл. 1 приведены технические характеристики рабочих мест, на которых производилось тестирование алгоритма.

Табл. 1. Сравнение характеристик тестовых рабочих мест

Рабочее место, № п/п	Процессор	Количество вычислителей, шт.
1	Intel Pentium 3805U	2
2	Intel Core i7-3770	8

3.2. Сравнение работы алгоритмов

Для определения эффективности работы алгоритма было программными методами отмечено время выполнения последовательной и параллельной версий алгоритма на четырех различных тестовых изображениях, приведенных на рис. 7.

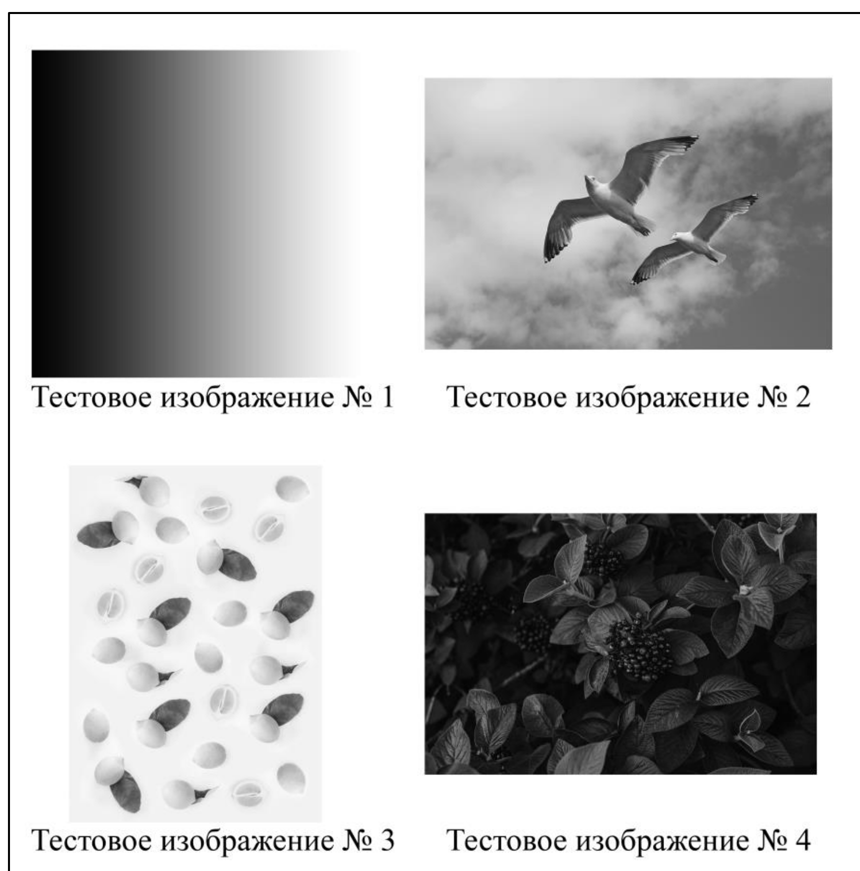

Рис. 7. Тестовые изображения

Каждое из изображений было поэтапно сжато с сохранением каждого этапа. Далее каждое из тестовых изображений было подано на вход программе в различных разрешениях.

В табл. 2 представлены результаты выполнения последовательной и параллельной версий алгоритма на тестовых рабочих местах № 1 и № 2 для различных размеров входного изображения. Помимо этого, результаты эксперимента также приведены в виде гистограмм на рис. 8, рис. 9.

Данные, полученные в ходе данного эксперимента, будут использоваться нами в дальнейшем для вычисления ускорения и эффективности разработанного алгоритма, как наиболее репрезентативные в плане размера по отношению к изображениям, предполагаемым к обработке алгоритмом.

Однако, в связи с высокой вариативность областей применения разработанной системы, размеры подаваемых на вход изображений также могут существенно отличаться. В связи с этим, было решено провести дополнительные тесты на изображениях крайне крупных и крайне малых изображениях. Данные эксперименты обеспечивают представление о работе алгоритма при различных исходных данных.

Табл. 2. Сравнение результатов выполнения алгоритма при различном размере входных данных на тестовом изображении № 1

Размер матрицы, эл.	Рабочее место № 1		Рабочее место № 2	
	Время работы последовательного алгоритма, с.	Время работы параллельного алгоритма, с.	Время работы последовательного алгоритма, с.	Время работы параллельного алгоритма, с.
94864	26,234	19,318	14,936	8,710
168100	48,350	35,217	27,338	14,939
299209	94,726	63,893	49,699	27,026
531441	169,518	115,742	88,886	48,788
944784	302,438	208,993	15,8457	86,200
1679616	540,030	372,870	286,415	153,835
2985984	972,356	665,509	521,413	276,107
5308416	1711,799	1186,050	857,886	560,453

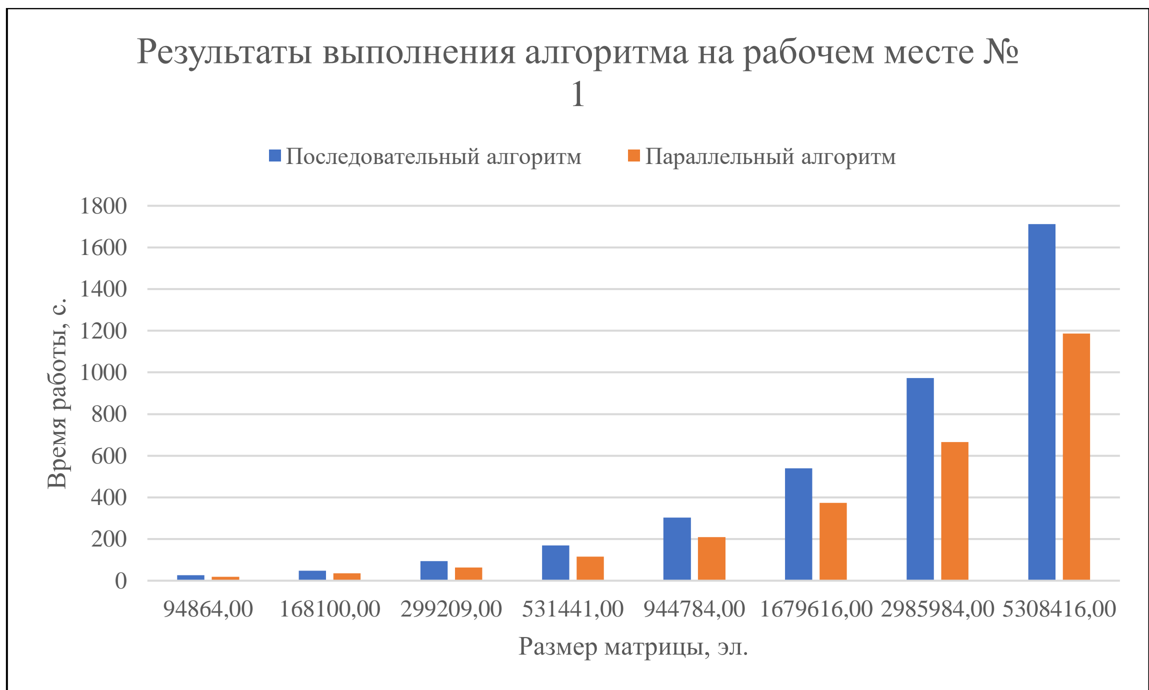

Рис. 8. Сравнение результатов выполнения алгоритма при различном размере входных данных на рабочем месте № 2

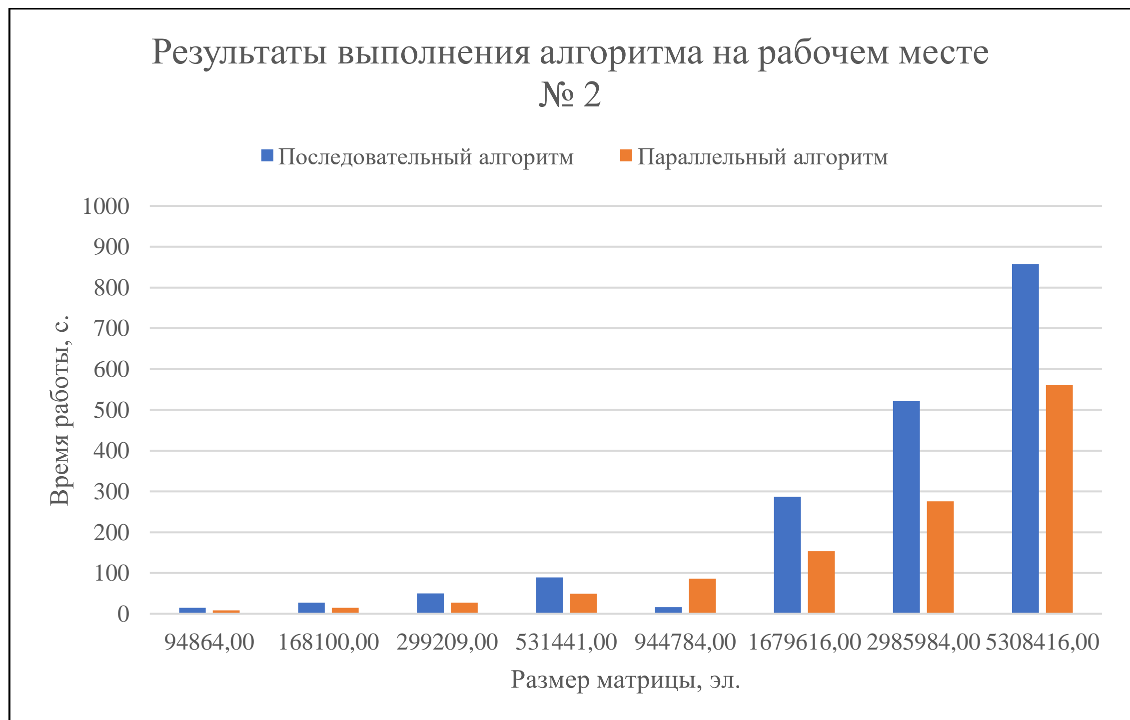

Рис. 9. Сравнение результатов выполнения алгоритма при различном размере входных данных на рабочем месте № 2

В табл. 3 представлены результаты выполнения последовательной и параллельной версий алгоритма на тестовых рабочих местах № 1 и № 2 для

крайне малых (231 в ширину и 231 высоту, и меньше) размеров входного изображения. Результаты эксперимента также приведены в виде графика на рис. 9.

Табл. 3. Сравнение результатов выполнения алгоритма при крайне малых размерах входных данных на тестовых изображения № 1-4

Разрешение исходного изображения	Рабочее место № 1		Рабочее место № 2	
	Время работы последовательного алгоритма, с.	Время работы параллельного алгоритма, с.	Время работы последовательного алгоритма, с.	Время работы параллельного алгоритма, с.
34,656	12,327	7,805	7,243	5,216
53,361	14,131	9,637	7,607	6,143
55,440	14,502	10,278	8,395	6,402

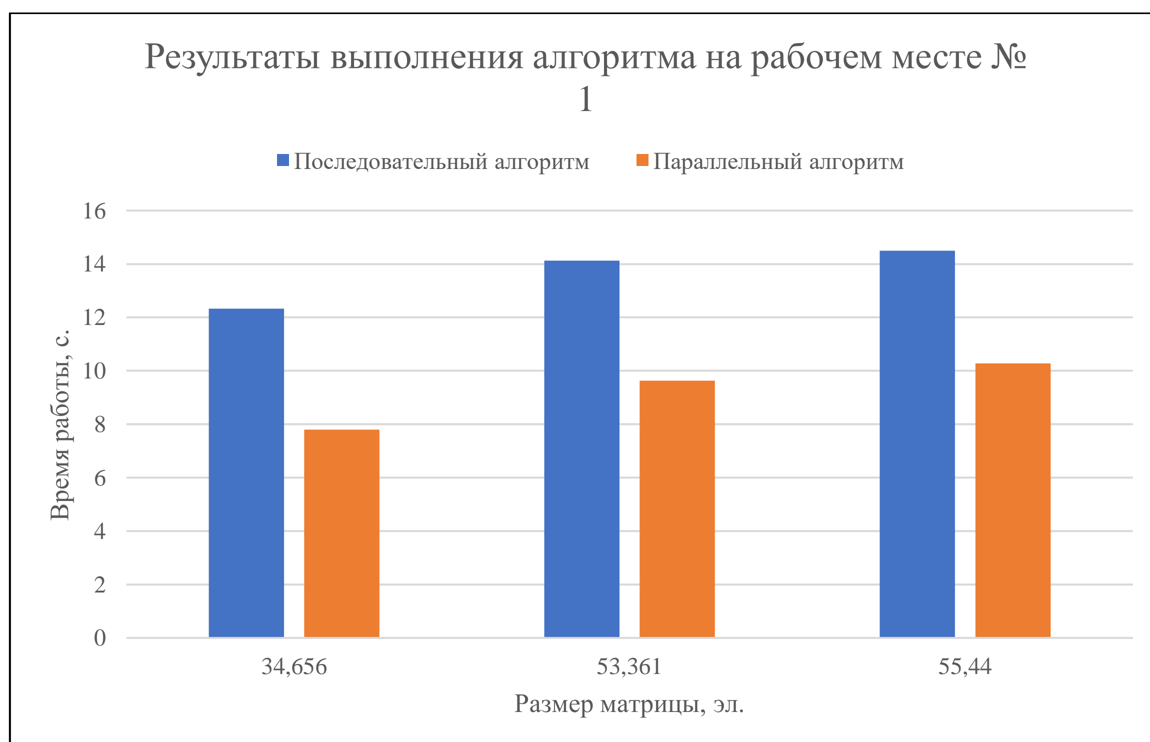

Рис. 10. Сравнение результатов выполнения алгоритма при различном размере малых входных данных на рабочем месте № 1

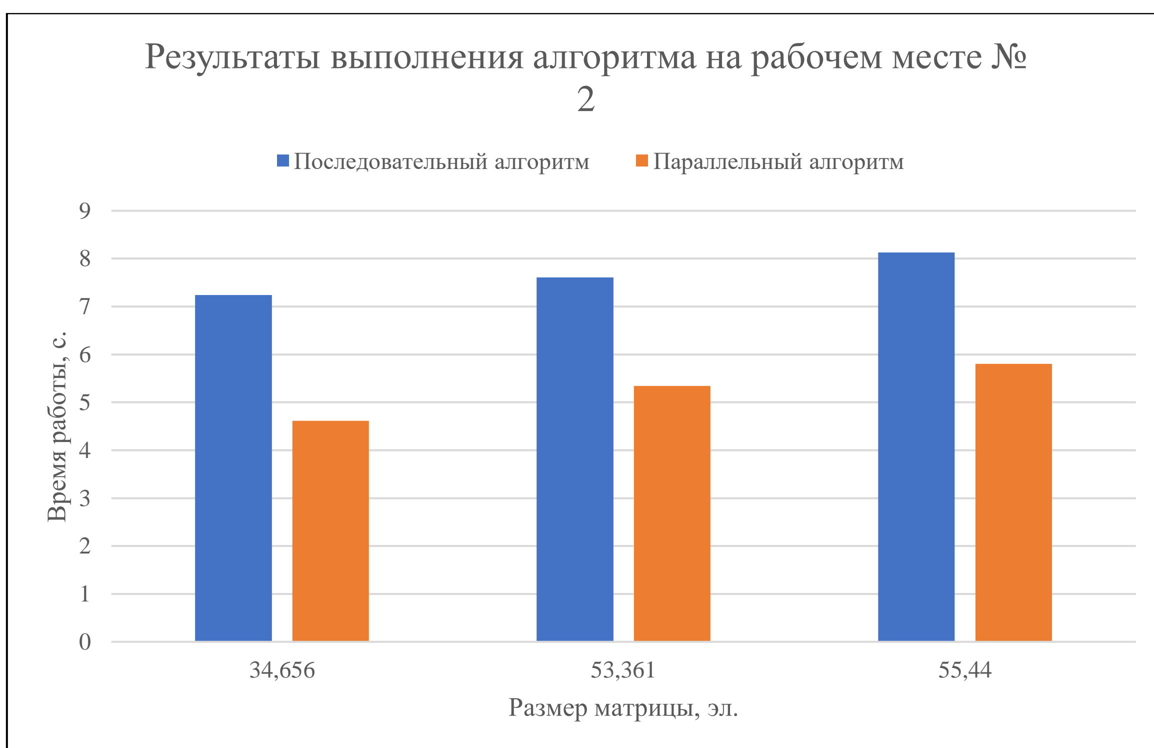

Рис. 9. Сравнение результатов выполнения алгоритма при различном размере малых входных данных на рабочем месте № 2

В табл. 4 представлены результаты выполнения последовательной и параллельной версий алгоритма на тестовых рабочих местах № 1 и № 2 для крайне больших (4096 в ширину и 4096 высоту, и больше) размеров входного изображения. Результаты эксперимента также приведены в виде графика на рис. 10.

Табл. 4. Сравнение результатов выполнения алгоритма при крайне больших размерах входных данных на тестовых изображениях № 1-4

Разрешение исходного изображения	Рабочее место № 1		Рабочее место № 2	
	Время работы последовательного алгоритма, с.	Время работы параллельного алгоритма, с.	Время работы последовательного алгоритма, с.	Время работы параллельного алгоритма, с.
10274836	971,926	900,364	828,706	762,338
12004482	1091,485	911,338	878,706	801,312

По итогам проведения данных экспериментов возможно рассчитать ускорение S и эффективность E данного алгоритма, что отражено в табл. 5.

Табл. 5. Вычисление ускорения и эффективности алгоритма для размера входной матрицы n = 531441 элементов

p	T_1	T_p	S	E
2	169,518	115,742	1,46	0,73
8	88,886	48,788	1,82	0,23

Вычисленные значения показывают значительное ускорение для компьютеров с двумя вычислителями, однако, в связи с накладными расходами, эффективность алгоритма при восьми вычислителях значительно снижается. Данные показатели говорят о необходимости дальнейшей оптимизации параллельного алгоритма с целью улучшения его масштабируемости.

Выводы по четвертой главе

В рамках данной главы было рассмотрено понятие вычислительного эксперимента, приведены исходные данные, на которых проводилось тестирование и проведены вычислительные эксперименты, направленные на изучение ускорения и эффективности данного алгоритма.

ЗАКЛЮЧЕНИЕ

В рамках данной работы был осуществлен обзор литературы и существующих решений, реализованы последовательная и параллельная версии алгоритма для вычисления линейной вариации, проведены численные эксперименты и оценены их результаты.

По итогам выполнения данных задач, поставленных ранее, можно сказать о выполнении и цели данной работы - разработки и реализации параллельного алгоритма для вычисления линейной вариации.

Дальнейшим направлением доработки должно стать улучшение масштабируемости, повышение эффективности и ускорения в работе реализованного алгоритма, создание специализированных программных инструментов, основанных на результатах данной работы.

ЛИТЕРАТУРА

1. Alexeev F. Linear Variation and Optimization of Algorithms for Connected Components Labeling in Binary Images. – Moscow, 2014. – 11 p.
2. Allen R., Hanson E. Computer Vision Systems. – Academic Press, 1978. – 389 p.
3. Chirag J. An Adaptive Parallel Algorithm for Computing. // Georgia Institute of Technology, 2017. – P. 13.
4. Chochia P., Milukova, O. Comparison of Two-Dimensional Variations in the Context of the Digital Image Complexity Assessment. // Journal of Communications Technology and Electronics, 2015. – P. 1432–1440.
5. Makovetskii A., Kober V. Image restoration based on topological properties of functions of two variables. // SPIE Applications of Digital image processing, 2012, – 12 p.
6. OpenCV. [Электронный ресурс] URL: https://docs.opencv.org/ (дата обращения: 16.02.2019)
7. OpenMP. [Электронный ресурс] URL: https://computing.llnl.gov/tutorials/openMP/ (дата обращения: 21.03.2019).
8. Shervin S. Camera as the instrument: the rising trend of vision based measurement. // IEEE Instrumentation & Measurement Magazine, 2014. – P. 41–47.
9. Wiggins R. Image File Formats: Past, Present and Future. // RadioGraphics, 2001. – P. 789–798.
10. Zierenberg J. Scaling properties of a parallel implementation of the multicanonical algorithm. // Computer Physics Communications, 2013. pp. 1155–1160.
11. Антонов А. Параллельное программирование с использованием OpenMP. – М.: Издательство московского университета, 2009. – 77 с.
12. Бодягин И. Эпоха параллельности. Способы выживания в эпоху многоядерного параллелизма. // RSDN Magazine, 2009. – с. 48.
13. Вирт Н. Алгоритмы и структуры данных. Цюрих, 1985. – 410 с.

14. Витушкин А. О многомерных вариациях. Москва: Государственное издательство технико-теоретической литературы, 1955. – 220 с.
15. Воеводин В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. – 608 с.
16. Ерош И. Дискретная математика. Учебное пособие для вузов. СПб.: СПбГУАП, 2005. – 144 с.
17. Оленев Н. Основы параллельного программирования в системе MPI. Москва: Вычислительный центр им. А. А. Дородницына, 2005. – 81 с.
18. Пальян Р. Эффективность и ускорение параллельных программ // Московский физико-технический институт. 2011. – с. 18.
19. Параллельные вычисления CUDA. [Электронный ресурс] URL: https://www.nvidia.ru/object/cuda-parallel-computing-ru.html (дата обращения: 16.04.2019).
20. Самарский А. Математическое моделирование и вычислительный эксперимент. // Вестник АН СССР, 1979. – с. 38–49.