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»

Отчет по лабораторной работе №4 по курсу "Анализ алгоритмов"

Тема Параллельные вычисления на основе нативных потоков

Студент Иммореева М.А.

Группа ИУ7-52Б

Оценка (баллы)

Преподаватель Строганов Д.В.

Москва — 2023 г.

Оглавление

Введение

Рост числа сложных задач, решение которых связано с применением современных информационных технологий, ведёт к необходимости использования параллельных вычислений. Параллельные вычисления носят междисциплинарный характер. Они затрагивают, в частности, такие области, как численные методы, структуры и алгоритмы обработки данных, аппаратное и программное обеспечение, системный анализ. Это позволяет применять знания, полученные при исследовании параллельных вычислений, в различных сферах научно-практической деятельности.

Параллельные вычисления — современная многогранная область вычислительных наук, бурно развивающаяся и являющаяся наиболее актуальной в ближайшие десятилетия. Актуальность данной области складывается из множества факторов, и в первую очередь, исходя из потребности в больших вычислительных ресурсах для решения прикладных задач моделирования процессов в физике, биофизике, химии и др. К тому же, традиционные последовательные архитектуры вычислителей и схем вычислений находятся на пороге технологического предела. В то же время технологический прорыв в области создания средств межпроцессорных и межкомпьютерных коммуникаций позволяет реализовать одно из ключевых звеньев параллелизма — эффективное управление в распределении вычислений по различным компонентам интегрированной вычислительной установки.

Целью данной работы является исследование и последующая реализация многопоточности на примере алгоритма Бойера–Мура поиска подстроки в строке (поиск одной и той же подстроки в файле от 100 Мбайт).

Для достижения поставленной цели необходимо решить следующие задачи:

1. исследовать алгоритм Бойера–Мура поиска подстроки в строке;
2. составить схему алгоритма;
3. определить средства программной реализации алгоритма;
4. реализовать алгоритм Бойера-Мура с использованием параллельных процессов и без;

5. протестировать разработанное программное обеспечение;

6. исследовать влияние количества потоков на время работы программы.

1 Аналитическая часть

1.1 Алгоритм Бойера–Мура поиска подстроки в строке

Алгоритм Бойера–Мура считается наиболее быстрым среди алгоритмов общего назначения, предназначенных для поиска подстроки в строке. Важной особенностью алгоритма является то, что он выполняет сравнения в шаблоне справа налево в отличие от многих других алгоритмов.

Алгоритм сравнивает символы шаблона x справа налево, начиная с самого правого, один за другим с символами исходной строки y. Если символы совпадают, производится сравнение предпоследнего символа шаблона и так до конца. Если все символы шаблона совпали с наложенными символами строки, значит, подстрока найдена, и поиск окончен. В случае несовпадения какого-либо символа (или полного совпадения всего шаблона) он использует две предварительно вычисляемые функции, чтобы сдвинуть позицию для начала сравнения вправо.

Таким образом для сдвига позиции начала сравнения алгоритм Бойера-Мура выбирает между двумя эвристическими функциями, называемыми эвристиками хорошего суффикса и плохого символа. Так как функции эвристические, то выбор между ними простой — ищется такое итоговое значение, чтобы не проверялось максимальное число позиций и при этом найдены все подстроки, равные шаблону. В итоге, эвристика стоп-символа – последнее вхождение стоп-символа в подстроку, эвристика безопасного (совпавшего) суффикса – для каждого возможного суффикса S шаблона указываем наименьшую величину, на которую нужно сдвинуть вправо шаблон, чтобы он снова совпал.

Если же какой-то символ образца не совпадает с соответствующим символом строки, образец сдвигается на величину сдвига вправо, и проверка снова начинается с последнего символа.

В качестве величины сдвига берется большее из двух значений: 1) Значение, полученное с помощью таблицы стоп-символов по простому правилу: Если несовпадение произошло на позиции i, а стоп-символ «с», то

значение величины сдвига будет равно $i - StopTable[c]$, где $StopTable[c]$ – значение, записанное в таблице стоп-символов, для символа «c». 2) Значение, полученное из таблицы суффиксов.

1.2 Параллельный алгоритм Бойера–Мура

Так как при работе алгоритма отступ после каждого сравнения может быть рассчитан только после нахождения несоответствия в тексте, был выбран подход разделения текста, в котором происходит поиск паттерна, на равные части. Вычисление результата для каждой части будет независим от вычисления результата других частей. При ситуации, где разделение текста происходит на паттерне, выбран подход, где параллельный процесс с собственной частью может получить доступ к следующей за ней части текста ровно на размер строки паттерна.

Вывод

В данном разделе был рассмотрен алгоритм Бойера–Мура и возможность его оптимизации с помощью распараллеливания потоков.

2 Конструкторская часть

В данном разделе приводится схема рассматриваемого алгоритма, определяются требования к программе.

2.1 Требования к программе

К вводу программы прилагаются данные требования: На вход подается название файла, строка, вхождения которой ищутся в тексте файла.

На выходе программа выводит индексы вхождений строки в тексте.

Требования к программе:

1. Корректное нахождение всех вхождений паттерна внутри текста;
2. Программа не должна аварийно завершаться при вводе неверного файла;
3. При вводе пустого паттерна программа не должна начинать поиск.

2.2 Разработка алгоритмов

На рисунке 2.1 приведена схема алгоритма Бойера–Мура поиска подстроки в строке. На рисунке 2.2 приведена схема подпрограммы поиска хорошего суффикса. На рисунке 2.3 приведена схема подпрограммы поиска плохого символа. На рисунке 2.4 приведена схема алгоритма Бойера–Мура поиска подстроки в строке для параллельных процессов. На рисунке 2.5 приведена схема главного потока (диспетчера) алгоритма Бойера–Мура поиска подстроки в строке.

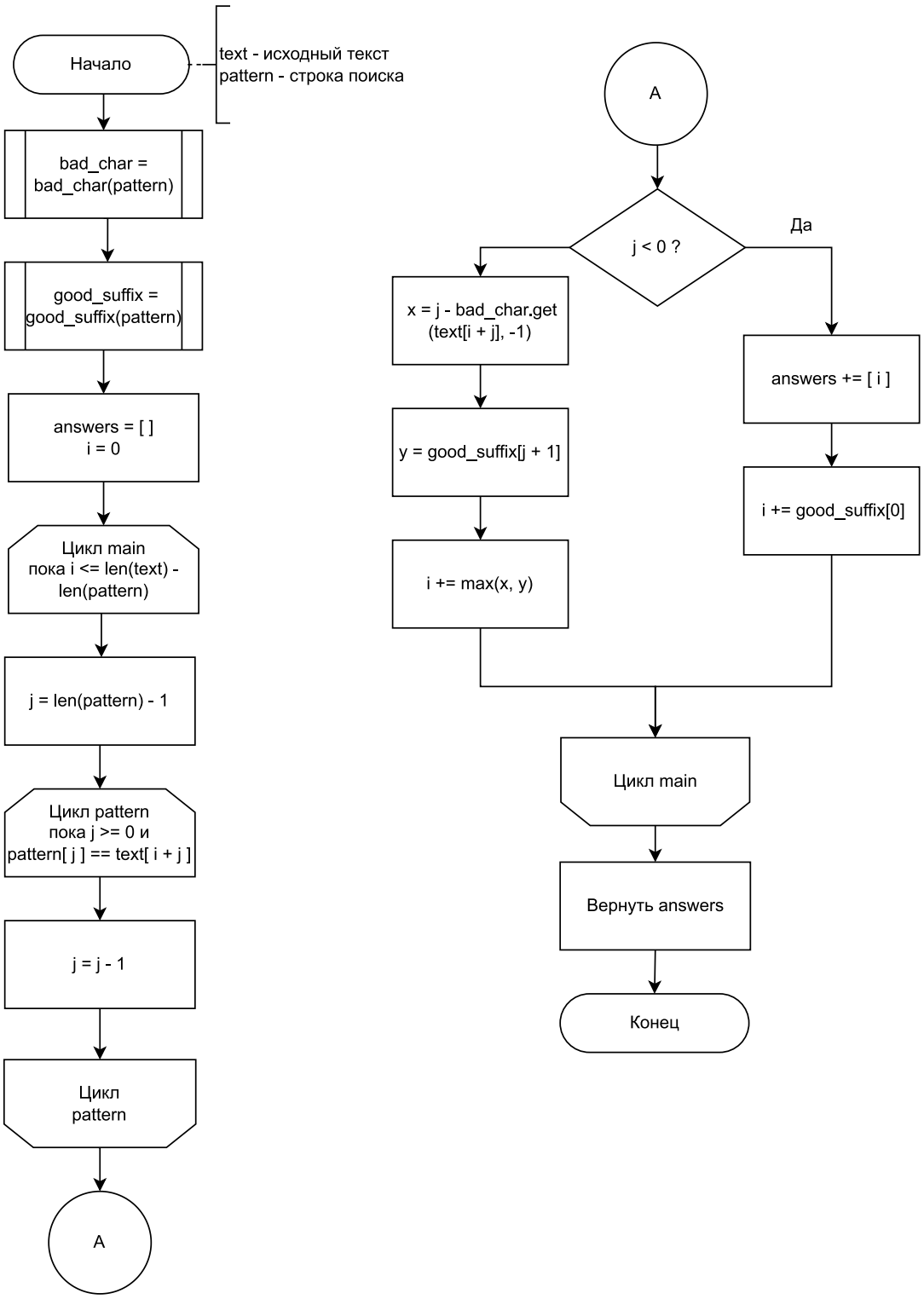

Рисунок 2.1 – Схема алгоритма Бойера–Мура поиска подстроки в строке

Вывод

В данном разделе была рассмотрена схема алгоритма Бойера–Мура, были приведены требования к программе.

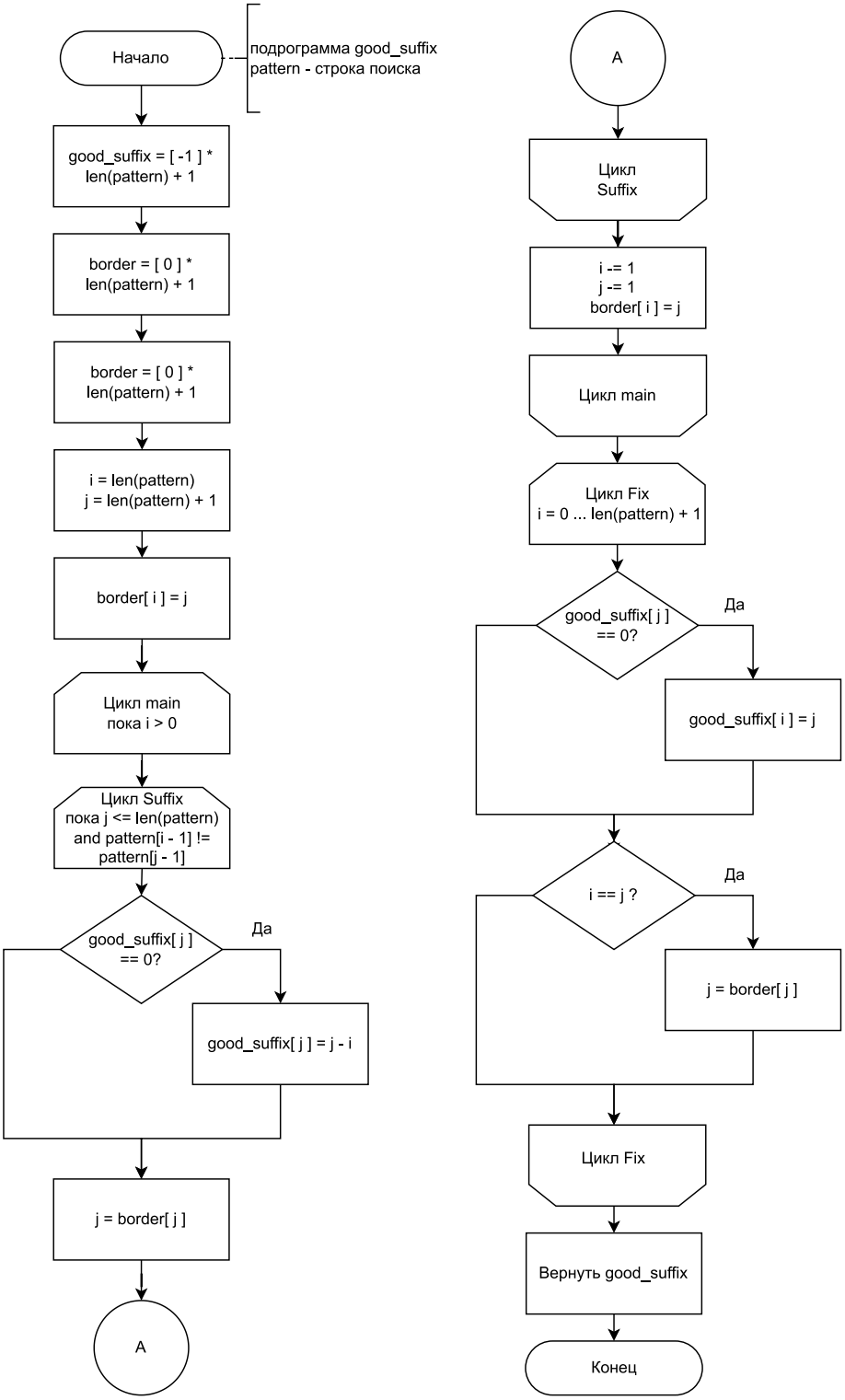

Рисунок 2.2 – Схема подпрограммы поиска хорошего суффикса

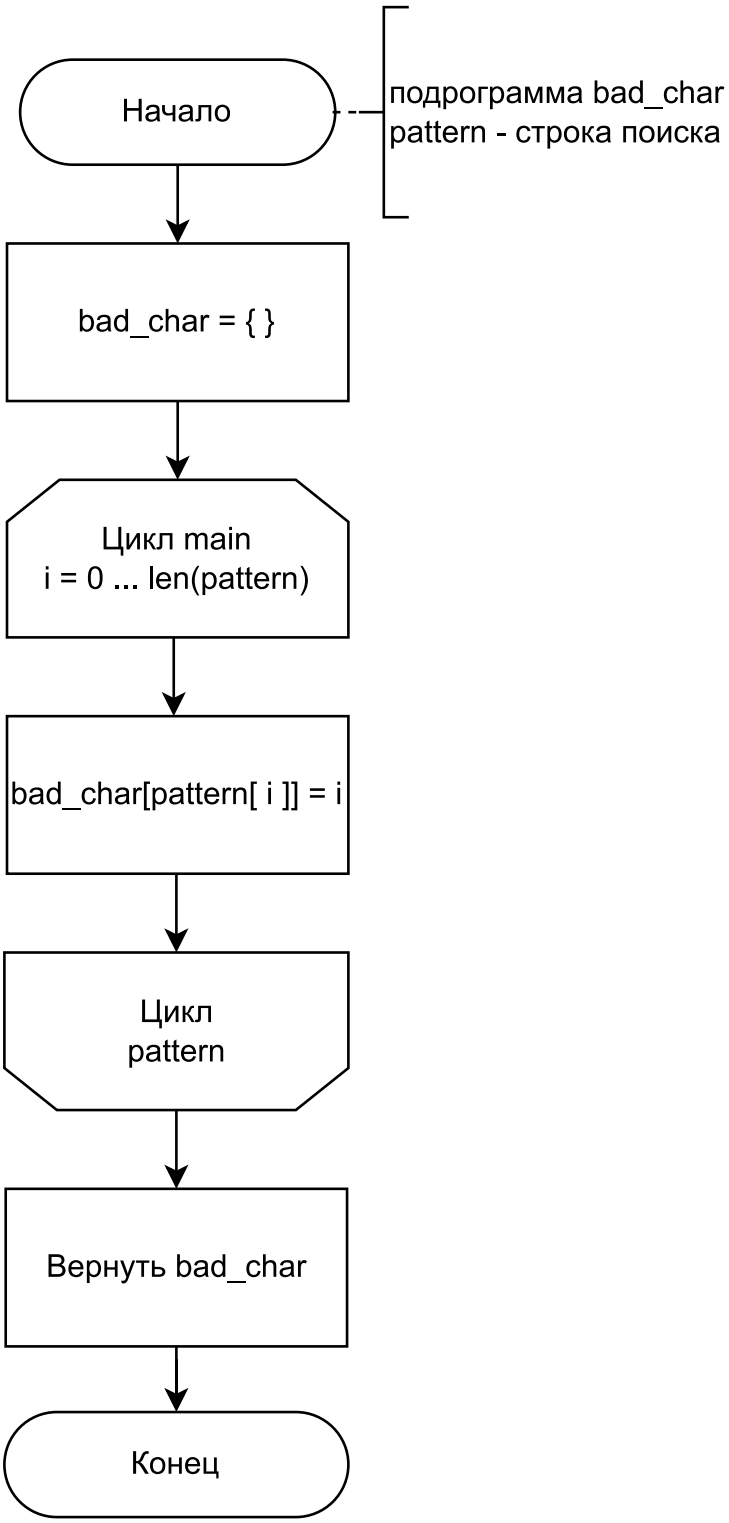

Рисунок 2.3 – Схема подпрограммы поиска плохого символа

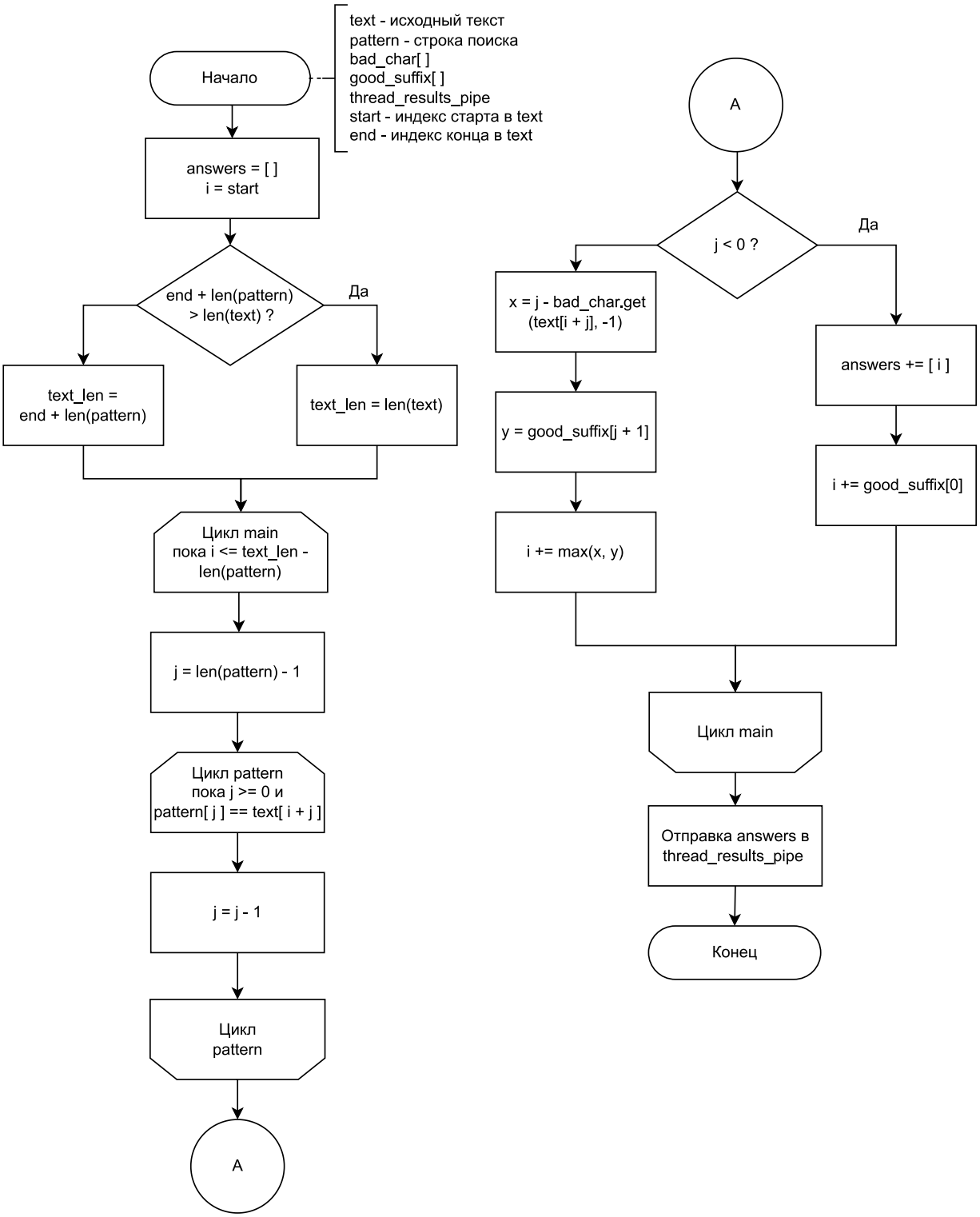

Рисунок 2.4 – Схема рабочего потока алгоритма Бойера–Мура поиска подстроки в строке

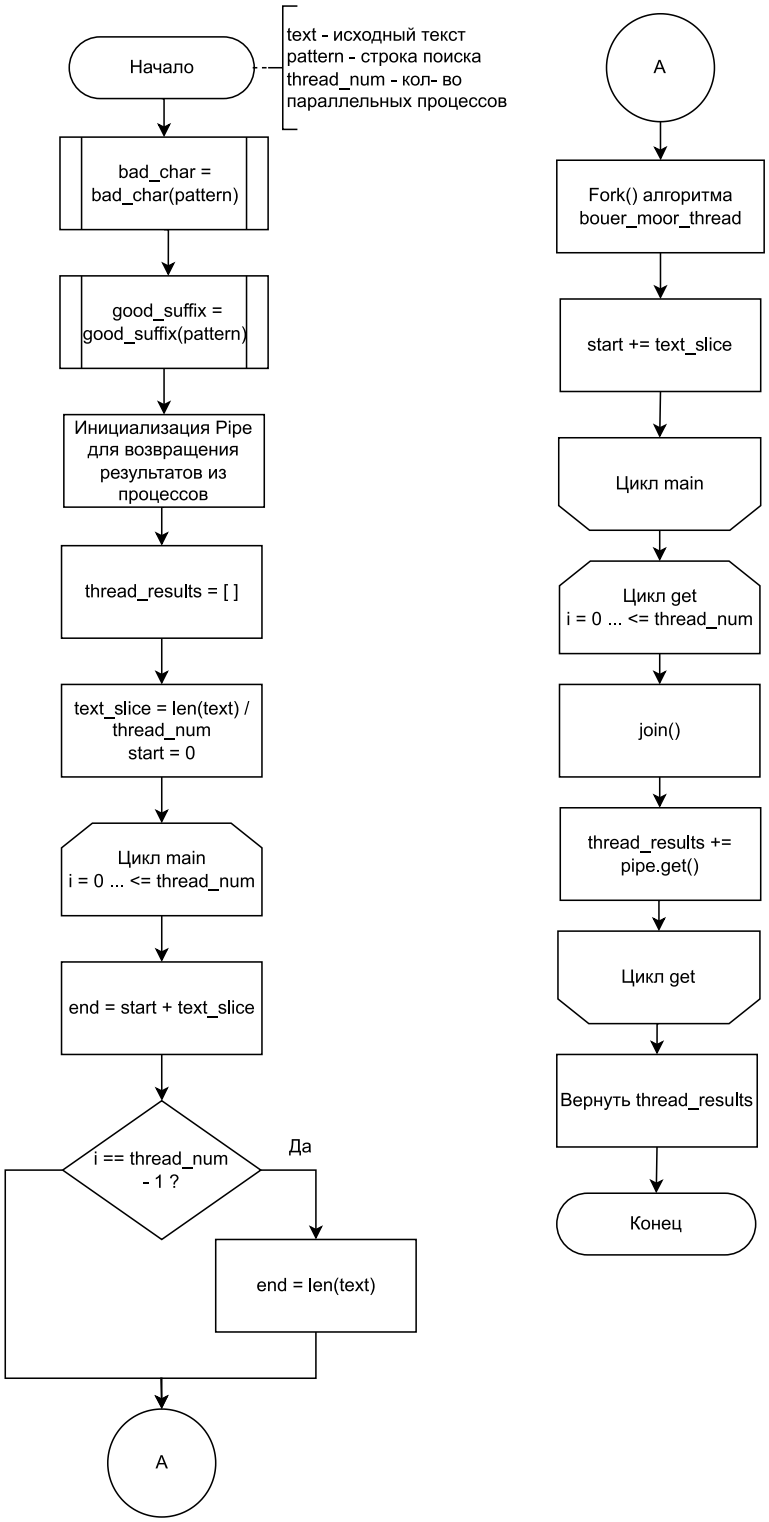

Рисунок 2.5 – Схема главного потока (диспетчера) алгоритма Бойера–Мура поиска подстроки в строке

3 Технологическая часть

В данном разделе будут приведены средства реализации и листинги кода.

3.1 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык программирования Python. Выбор обусловлен наличием библиотек для измерения времени, наличием инструментов для работы с параллельными потоками. Были использованы библиотеки threading, multiprocessing, os.

3.2 Сведения о модулях программы

Программа состоит из следующих модулей: main.py - главный файл программы, в котором располагается вся программа, timer.py - файл программы с замерами времени.

3.3 Реализация алгоритмов

В листингах 3.1, 3.2, 3.3, 3.4, 3.5, приведены реализации алгоритмов.

Листинг 3.1 – Алгоритм Бойера–Мура поиска подстроки в строке.

```
    def bouer_moor(text, pattern):
        bad_char = bad_char_heuristic(pattern)
        good_suffix = good_suffix_heuristic(pattern)
        i = 0
        indexes = []
        while i <= len(text) - len(pattern):
            j = len(pattern) - 1
            while j >= 0 and pattern[j] == text[i + j]:
                j -= 1
            if j < 0:
                indexes.append(i)
                i += good_suffix[0]
            else:
                x = j - bad_char.get(text[i + j], -1)
                y = good_suffix[j + 1]
                i += max(x, y)
        return indexes
```

Листинг 3.2 – Эвристика хорошего суффикса

```
1     def good_suffix_heuristic(pattern):
2         good_suffix = [-1] * (len(pattern) + 1)
3         border = [0] * (len(pattern) + 1)
4         i = len(pattern)
5         j = len(pattern) + 1
6         border[i] = j
7         while i > 0:
8             while j <= len(pattern) and pattern[i - 1] !=
                  pattern[j - 1]:
9                 if good_suffix[j] == 0:
10                    good_suffix[j] = j - i
11            j = border[j]
12            i -= 1
13            j -= 1
14        border[i] = j
15        j = border[0]
16        for i in range(len(pattern) + 1):
17            if good_suffix[i] == -1:
18                good_suffix[i] = j
19            if i == j:
20                j = border[j]
21    return good_suffix
```

Листинг 3.3 – Эвристика плохого символа

```
1     def bad_char_heuristic(pattern):
2         bad_char = {}
3         for i in range(len(pattern)):
4             bad_char[pattern[i]] = i
5         return bad_char
```

Листинг 3.4 – Схема рабочего потока алгоритм Бойера–Мура

```
def bouer_moor_thread(thread_results_pipe, text, pattern,
    good_suffix, bad_char, start, end):
    indexes = []
    pattern_len = len(pattern)
    i = start
    text_len = len(text) if end + pattern_len > len(text) else
        end + pattern_len

    while i <= text_len - pattern_len:
        j = pattern_len - 1
        while j >= 0 and pattern[j] == text[i + j]:
            j -= 1
        if j < 0:
            indexes.append(i)
            i += good_suffix[0]
        else:
            x = j - bad_char.get(text[i + j], -1)
            y = good_suffix[j + 1]
        if (max(x, y) > text_len - pattern_len and max(x, y) <
            text_len):
            i = text_len - pattern_len
        else:
            i += max(x, y)
    thread_results_pipe.send(indexes)
```

Листинг 3.5 – Схема главного потока (диспетчера) алгоритма Бойера–Мура

```
def bouer_moor_parallel(text, pattern, thread_num):
    thread_results_pipe,thread_results_pipe_recieve = Pipe()
    thread_results = []
    bad_char = bad_char_heuristic(pattern)
    good_suffix = good_suffix_heuristic(pattern)
    text_len = len(text)

    text_slice = int(text_len / thread_num)

    k = 0

    threads = []
    for i in range(thread_num):
        k_end = k + text_slice
        if (i == thread_num - 1):
            k_end = text_len
        threads.append(threading.Thread(target=bouer_moor_thread,
           args=(thread_results_pipe, text, pattern, \
                        good_suffix, bad_char, k, k_end,)))
        k += text_slice
    for thread in threads:
        thread.start()

    for thread in threads:
        thread_results.append(thread_results_pipe_recieve.recv())
        thread.join()

    return thread_results
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритма Бойера–Мура.

Алгоритм прошел проверку с последовательным и параллельным запуском.

Таблица 3.1 – Тестирование функций

Строка	Паттерн	Ожидаемый результат
$abcdefabcdef$	abc	$0, 6$
abc	abc	0
$bcdeabcfabcdef$	abc	6
$bcdeacfabcdef$	abc	Пустой вывод
$bcdeacfabcdef$	Пустой ввод	ERR: pattern cant be empty
$hellowearetest3ing\_hellahell$	hel	$0, 29$

Вывод

Были разработан и протестирован алгоритм Бойера–Мура.

4 Исследовательская часть

В данном разделе будет приведена постановка замера времени и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялся замер времени:

- Операционная система: 64-разрядная операционная система, процессор x64.
- Память: 16 Гб.
- Процессор: Intel(R) Core(TM) i7–4700HQ CPU @ 2.40 ГГц.

Во время замера времени ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.2 Время выполнения алгоритмов

Алгоритмы замерялись при помощи функции $time.perf\_counter()$ из библиотеки *time* языка *Python*. Данная функция возвращает значение в долях секунды счетчика производительности, то есть часов с наибольшим доступным разрешением для измерения короткой длительности.

Замеры времени для каждого выполнения проводились 10 раз. В качестве результата взято среднее время работы алгоритма.

Как видно из замеров, эффективней всего работает алгоритм, использующий 16 потоков.

При рассмотрении работы алгоритма с 16, 32 и 64 потоками на рисунке 4.1 становится заметно, что разница между затраченным временем между 16 потоками и 32, 64 не превышает 0.2 секунд. Разница в работе алгоритма с 32 и 64 потоками на рисунке 4.2 не превышает 0.05 секунд. Также

стоит учесть, что одновременное выполнение большого количества потоков может тратить много процессорного времени на переключение контекста за счет фактической обработки, сбрасывая блоки памяти в файловую систему и увеличивая количество операций ввода/вывода, что в конечном итоге замедляет работу всего приложения и засоряет хост-машину[4].

Также возникает проблема в работе с памятью. Все потоки используют одну и ту же память процесса, но если каждый поток требует больше памяти, потоки будут ограничены памятью процесса[4].

Для данного алгоритма в предоставленной системе оптимальней всего будет использовать 16-поточный алгоритм.

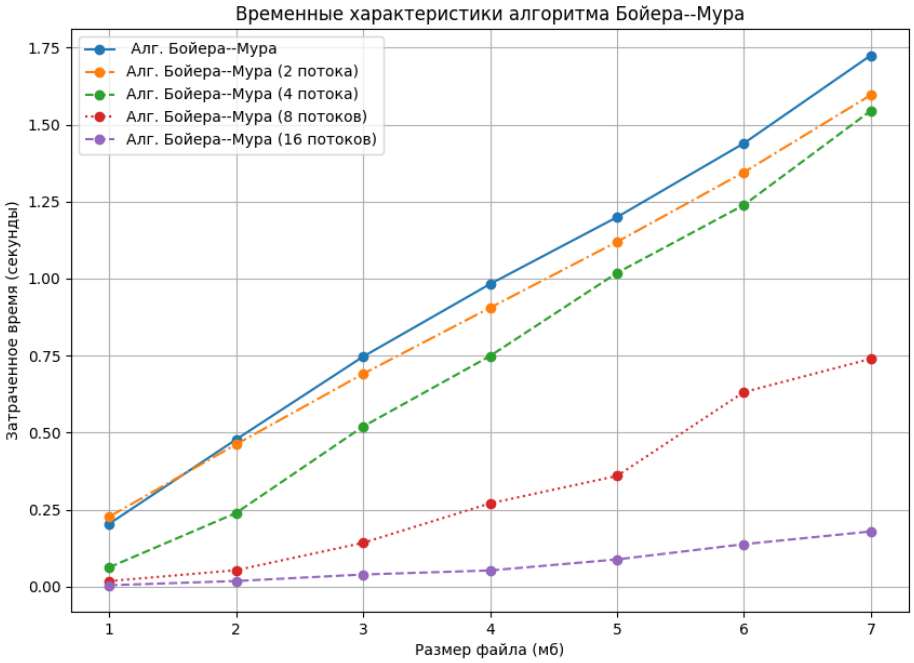

Рисунок 4.1 – Замеры времени для последовательного алгоритма Бойера–Мура в сравнении с алгоритмом, использующим потоки

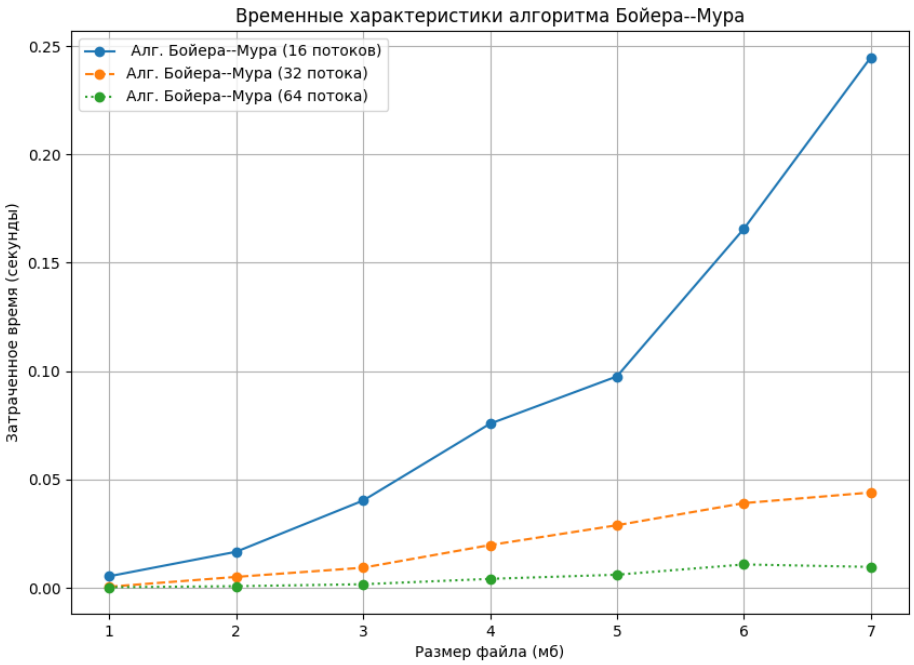

Рисунок 4.2 – Замеры времени для последовательного алгоритма Бойера–Мура в сравнении с алгоритмом, использующим потоки

Вывод

Из проведенного анализа можно сделать следующие выводы: использование многопоточности дает существенный прирост эффективности. Однако, при использовании большего количества потоков(более 16), прирост эффективности падает до несущественных размеров. Для устройства, на котором проводился эксперимент, оптимальное количество потоков для эффективной работы был выбран 16-поточный алгоритм.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

1. исследован алгоритм Бойера–Мура поиска подстроки в строке;
2. составлены схемы алгоритма Бойера–Мура, параллельного алгоритма Бойера–Мура;
3. определены средства программной реализации алгоритма;
4. реализован алгоритм Бойера-Мура с использованием параллельных процессов и без;
5. протестировано разработанное программное обеспечение;
6. исследовано влияние количества потоков на время работы программы.

Использование многопоточности дает существенный прирост эффективности. Однако, при использовании большего количества потоков(более 16), прирост эффективности падает до несущественных размеров. Последующее увеличение потоков нецелесообразно.

Стоит учесть, что одновременное выполнение большого количества потоков может тратить много процессорного времени на переключение контекста за счет фактической обработки, сбрасывая блоки памяти в файловую систему и увеличивая количество операций ввода/вывода, что в конечном итоге замедляет работу всего приложения и засоряет хост-машину[4].

Также возникает проблема в работе с памятью. Все потоки используют одну и ту же память процесса, но если каждый поток требует больше памяти, потоки будут ограничены памятью процесса[4].

Для устройства, на котором проводился эксперимент, оптимальное количество потоков для эффективной работы был выбран 16-поточный алгоритм.

Список использованных источников

[1] Волосова А. В. Параллельные методы и алгоритмы. – Москва: МАДИ, 2020.

[2] time — Time access and conversions // Python.org URL: https://docs.python.org/3/library/time.html (дата обращения: 12.11.2023).

[3] Definitive Guide: Threading in Python Tutorial, May 2020 URL: https://www.datacamp.com/tutorial/threading-in-python (дата обращения: 12.11.2023).

[4] Многопоточность и многопроцессорность – выбор правильного подхода для вашей разработки // Дори Экстерман, 6 октября 2020 г. URL: https://web-control.ru/novosti/news_post/mnogopotochnost-i-mnogoprocessornost-vybor-pravilnogo-podhoda-dlya-vashej-razrabotki (дата обращения: 13.11.2023).