



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 Фам Минь Хиеу

Группа ИУ7-52Б

Оценка (баллы)

Преподаватель Волкова Л.Л., Строганов Д.В.

Содержание

Введение	3
1 Аналитическая часть	4
1.1 Последовательный алгоритм Кнута-Морриса-Пратта	4
1.2 Параллельный алгоритм Кнута-Морриса-Пратта	5
2 Конструкторская часть	6
2.1 Разработка алгоритмов	6
3 Технологическая часть	15
3.1 Требования к программному обеспечению	15
3.2 Средства реализации	15
3.3 Сведения о модулях программы	15
3.4 Реализация алгоритмов	16
3.5 Функциональные тесты	18
4 Исследовательская часть	20
4.1 Технические характеристики	20
4.2 Демонстрация работы программы	20
4.3 Время выполнения реализаций алгоритмов	21
4.4 Вывод	24
Заключение	25
Список использованных источников	26

Введение

Существуют различные способы написания программ, одним из которых является использование параллельных вычислений. Параллельные вычисления — способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно [1].

Основная цель параллельных вычислений — уменьшение времени решения задачи. Многие задачи требуется решать в реальном времени или для их решения требуется очень большой объем вычислений. Таким трудоемким алгоритмом является, например, алгоритм Кнута-Морриса-Пратта — один из алгоритмов поиска подстроки в строке.

Целью данной лабораторной работы является изучение организации параллельных вычислений на базе алгоритма Кнута-Морриса-Пратта.

Для достижения поставленной цели требуется выполнить следующие задачи:

- описать последовательный и параллельный алгоритмы поиска подстроки в строке Кнута-Морриса-Пратта;
- построить схемы данных алгоритмов;
- создать программное обеспечение, реализующее рассматриваемые алгоритмы;
- провести сравнительный анализ по времени для реализованного алгоритма;
- подготовить отчет о выполненной лабораторной работе.

1 Аналитическая часть

В данном разделе будут представлены описания последовательного и параллельного вариантов алгоритма Кнута-Морриса-Пратта.

1.1 Последовательный алгоритм Кнута-Морриса-Пратта

Алгоритм Кнута-Морриса-Пратта (КМП) предназначен для эффективного поиска подстроки в строке. Его ключевая идея заключается в использовании информации о структуре самой подстроки для оптимизации процесса поиска [2].

Принцип его работы следующий:

- алгоритм начинает с создания массива lps , где каждый элемент $lps[i]$ представляет собой длину максимального собственного суффикса подстроки $s[0:i]$, который также является её префиксом;
- этот массив lps вычисляется для подстроки и используется для определения возможных сдвигов в случае несовпадения символов в процессе поиска;
- алгоритм сравнивает символы строки s и подстроки p поочередно;
- при несовпадении символов, если мы находимся в середине сравнения, используем значение lps для определения новой позиции в подстроке, с которой продолжим сравнение;
- это позволяет избежать бесполезных сравнений, так как мы используем предварительно вычисленные значения lps для оптимизации процесса;
- повторяем процесс сравнения до тех пор, пока не найдена подстрока или не достигнут конец строки; если найдена подстрока, добавляем индекс её вхождения в результирующий массив.

1.2 Параллельный алгоритм Кнута-Морриса-Пратта

Поскольку можно разбить строки на подстроки и обрабатывать их отдельно, то логично назначить каждому потоку часть строки, в которой нужно поискать подстроку [3].

Основная проблема этой версии заключается в том, что если паттерн поступает в часть разделения данных или точку соединения, он не обнаруживается, поскольку данные обрабатываются на разных потоках. Для решения этой проблемы мы обрабатываем еще часть строки в каждой точке соединения. При этом необходимо использовать 2 мьютекса в теле назначенного потока для организации монопольного доступа к массиву.

Вывод

Были изучены алгоритмы поиска подстроки в строке Кнута-Морриса-Пратта и его параллельная версия.

2 Конструкторская часть

В данном разделе будут представлены схемы алгоритмов поиска подстроки в строке: последовательного алгоритма Кнута-Морриса-Пратта, параллельного алгоритма Кнута-Морриса-Пратта.

2.1 Разработка алгоритмов

На рисунках 2.1–2.3 представлена схема последовательного алгоритма Кнута-Морриса-Пратта. Схема параллельного алгоритма Кнута-Морриса-Пратта представлена на рисунках 2.4–2.5. Схема алгоритма задачи одного потока представлена на рисунках 2.6–2.8.

Фан Минь Хуэй ЦУТ-525

1. Вай

Задача

Алгоритм КМП поиска подстроки в строке

При решении нужно воспользоваться мультитекст для взаимо-
исключения — при доступе к какому ресурсу?

Последовательный алгоритм КМП:

и массиву.

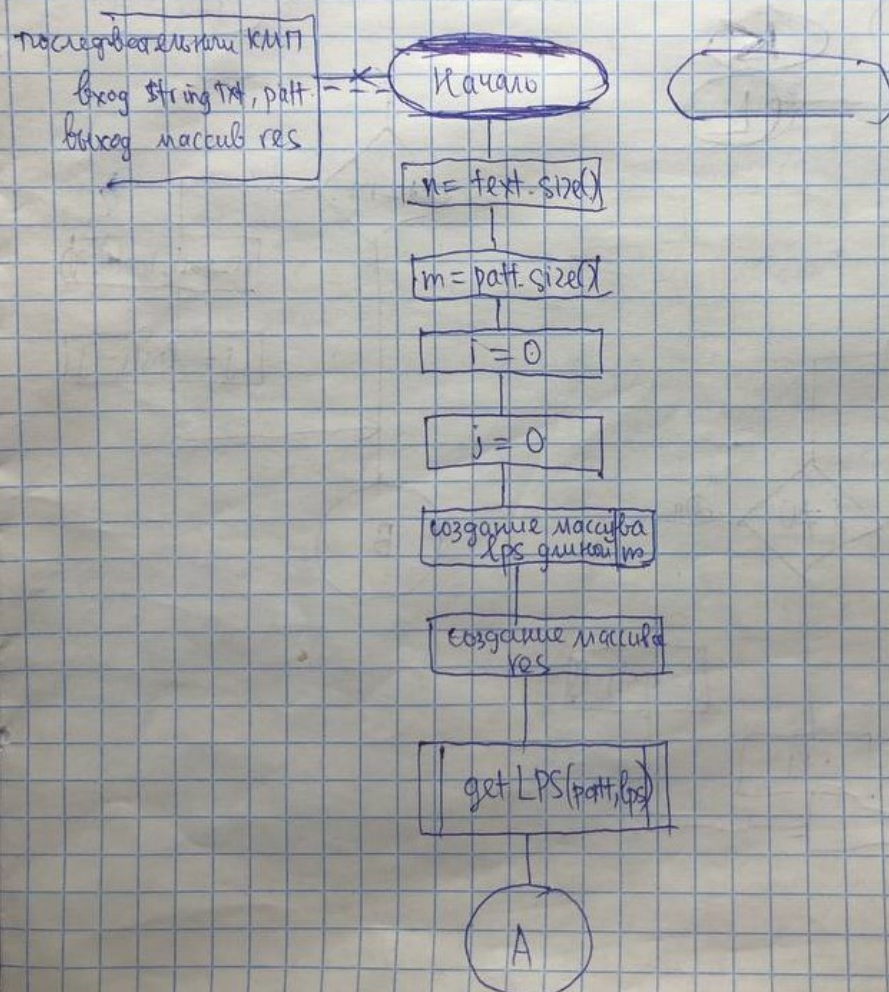


Рисунок 2.1 – Алгоритм последовательного поиска подстроки в строке КМП

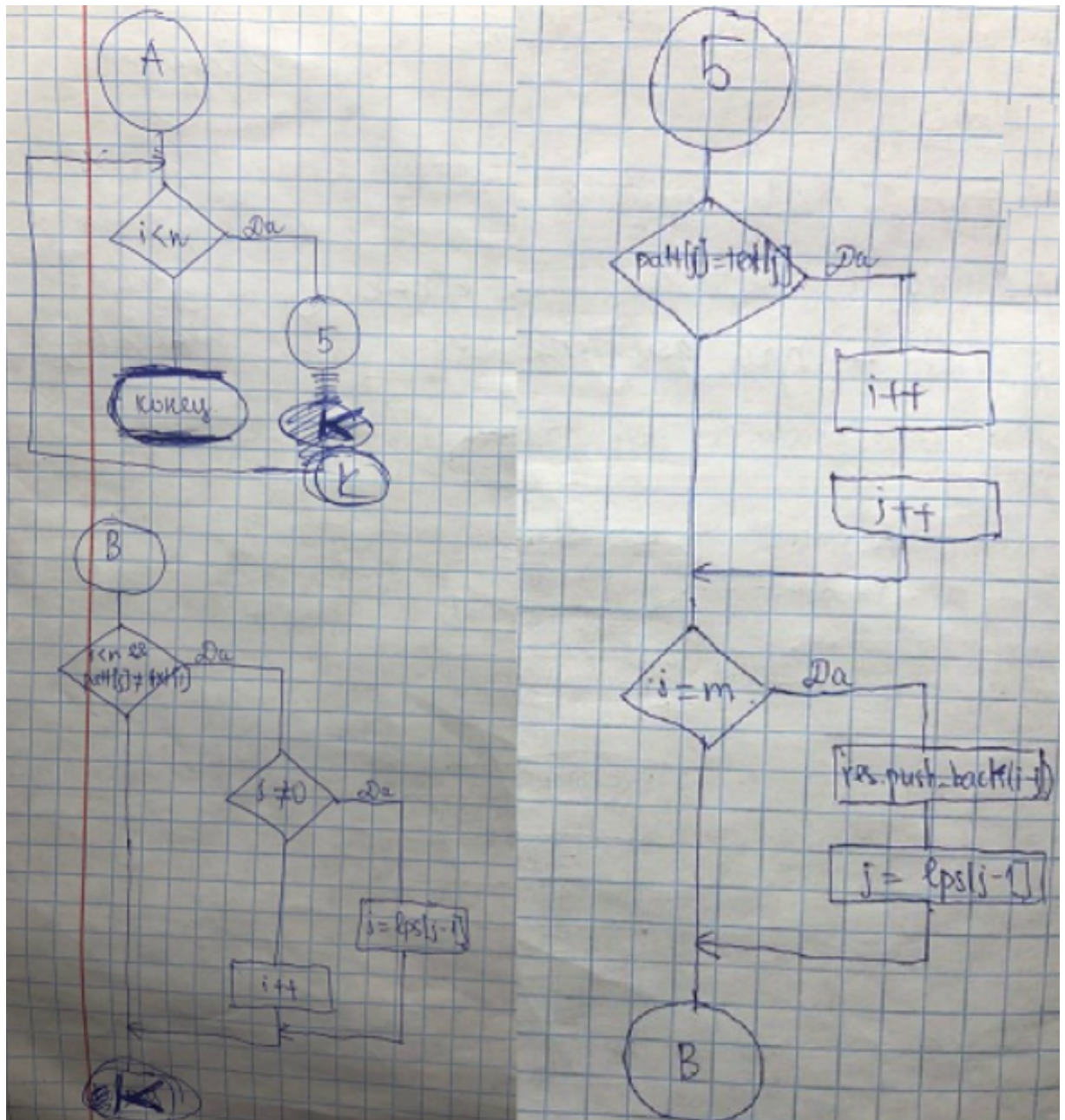


Рисунок 2.2 – Продолжение алгоритма последовательного поиска подстроки в строке КМП

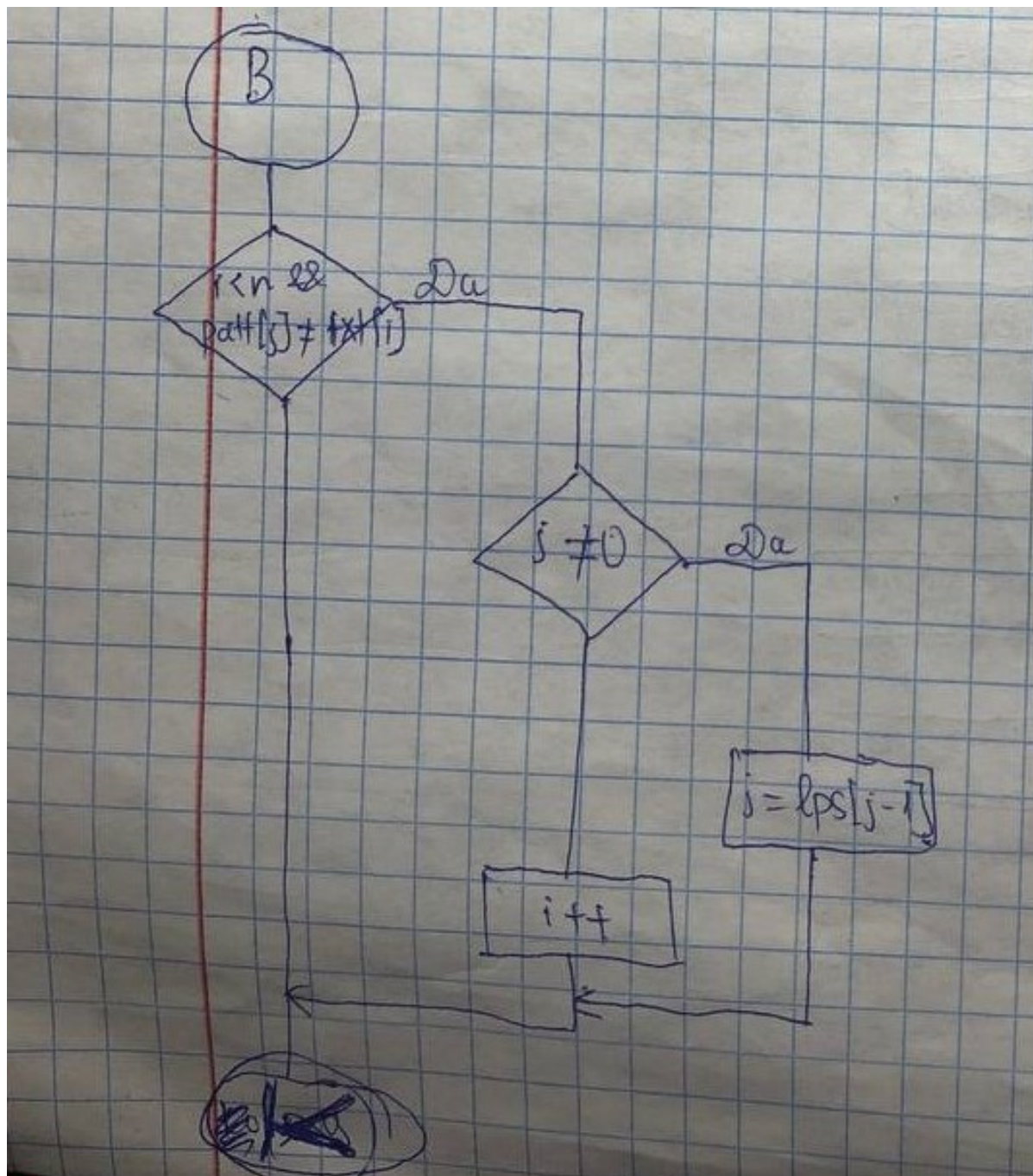


Рисунок 2.3 – Продолжение алгоритма последовательного поиска подстроки в строке КМП

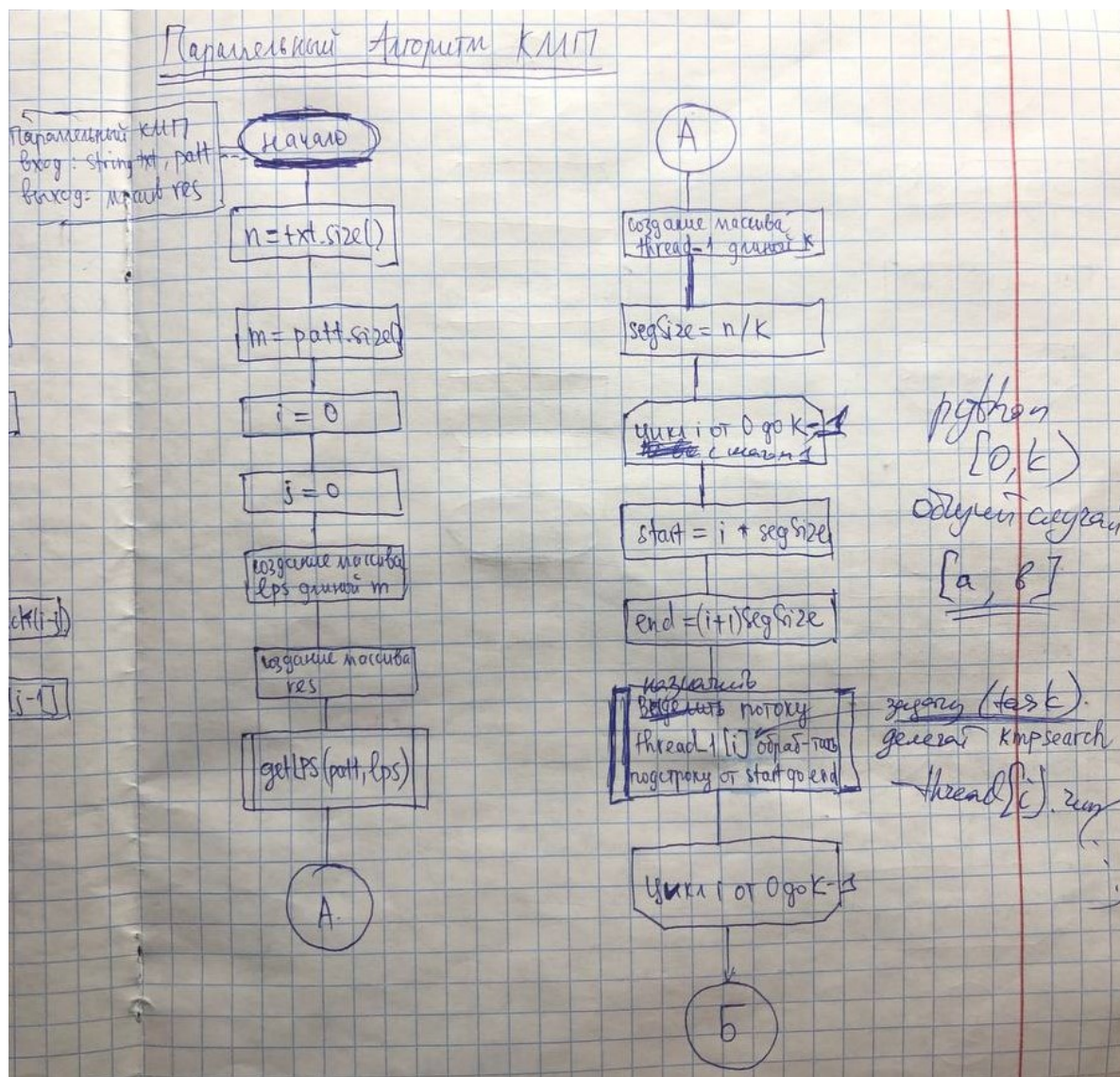


Рисунок 2.4 – Алгоритм параллельного поиска подстроки в строке КМП

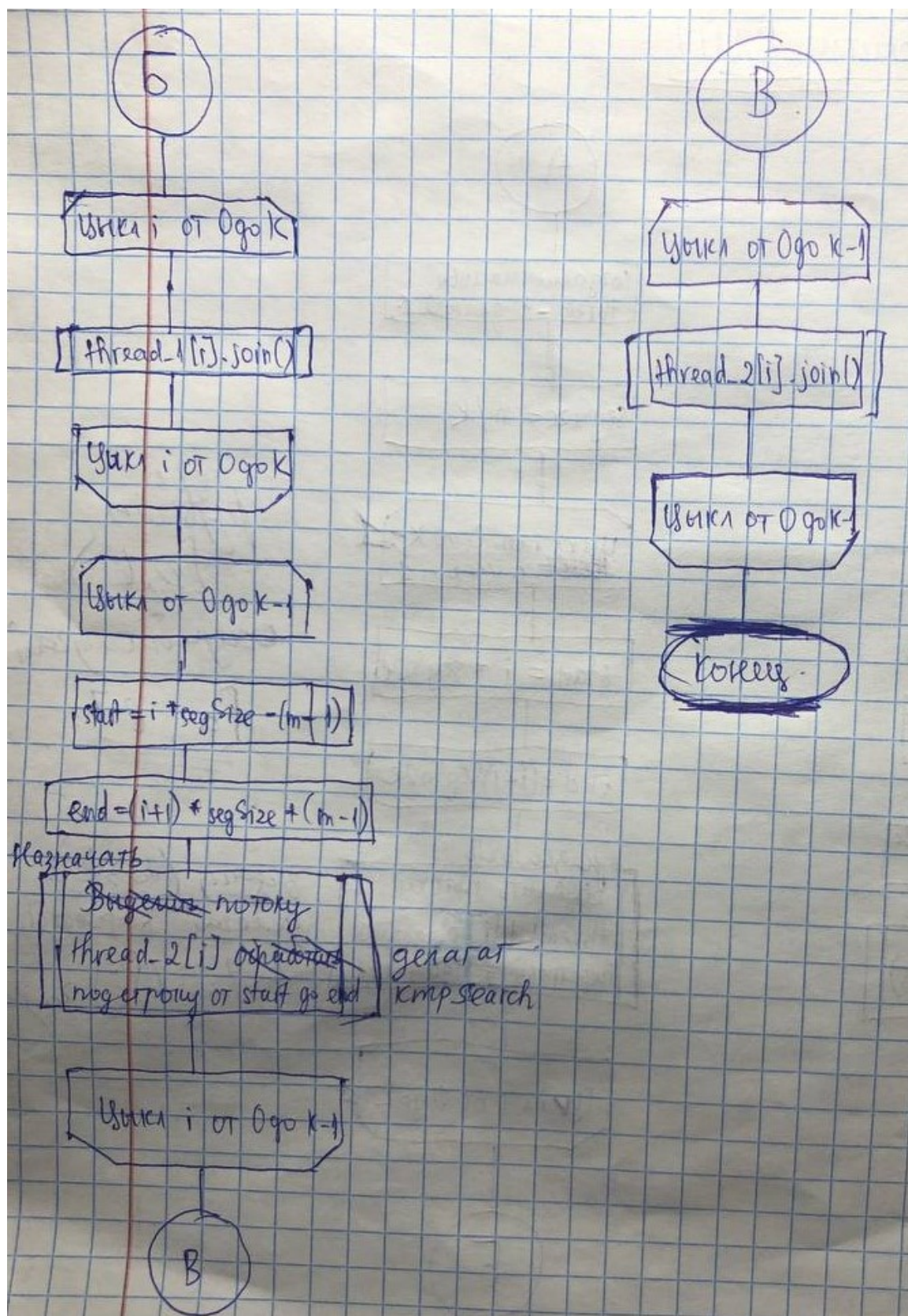


Рисунок 2.5 – Продолжение алгоритма параллельного поиска подстроки в строке КМП

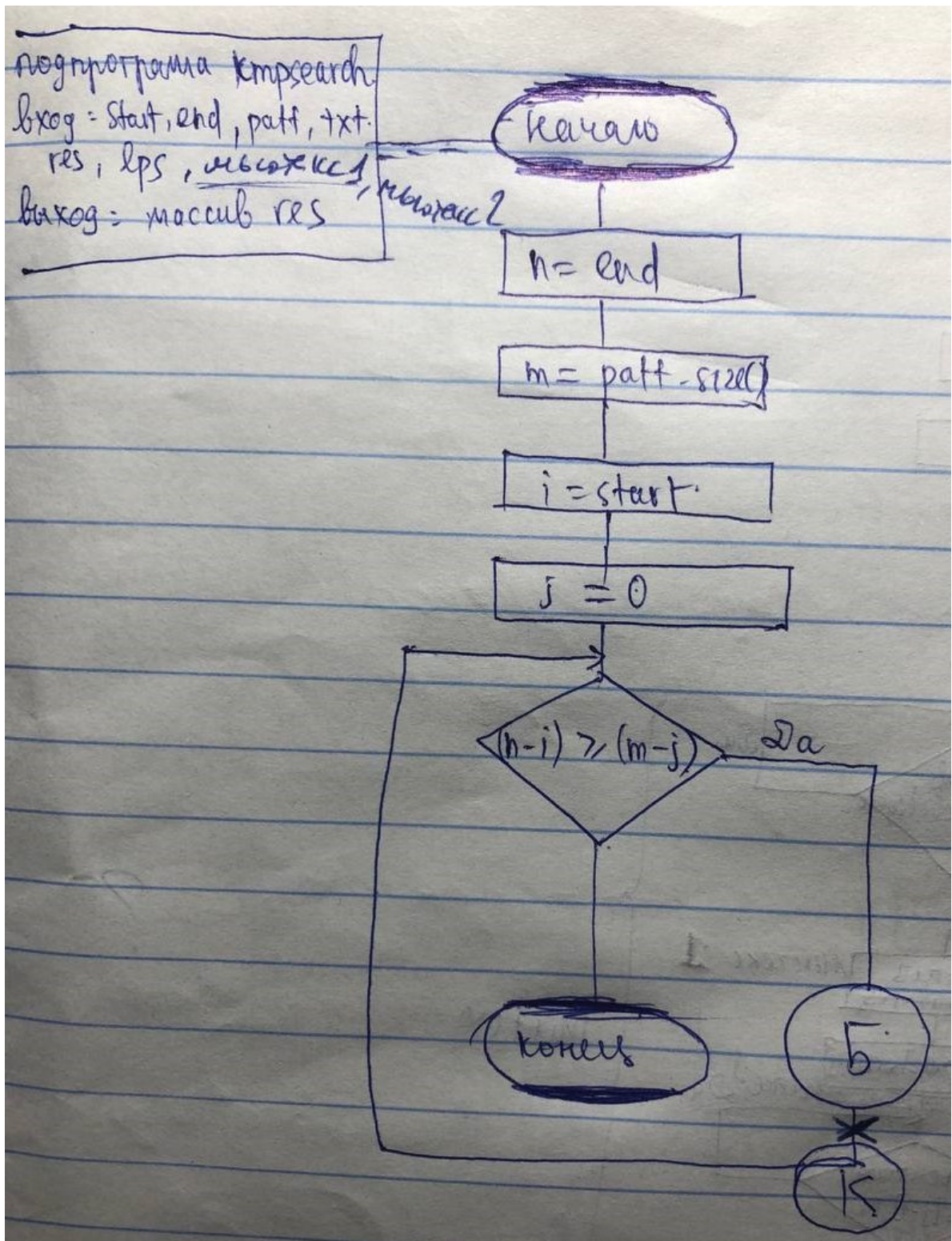


Рисунок 2.6 – Алгоритм задачи одного потока

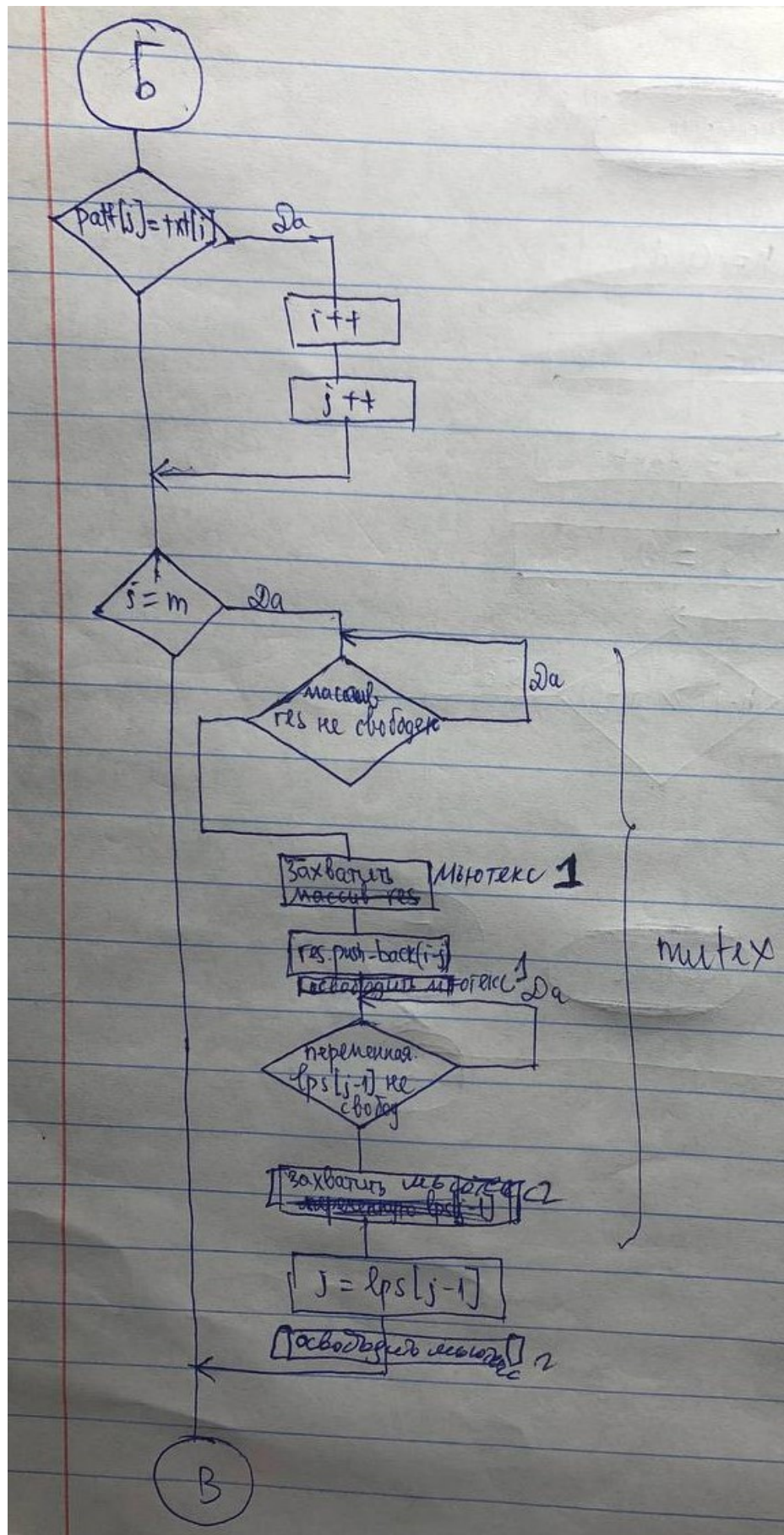


Рисунок 2.7 – Продолжение алгоритма задачи одного потока

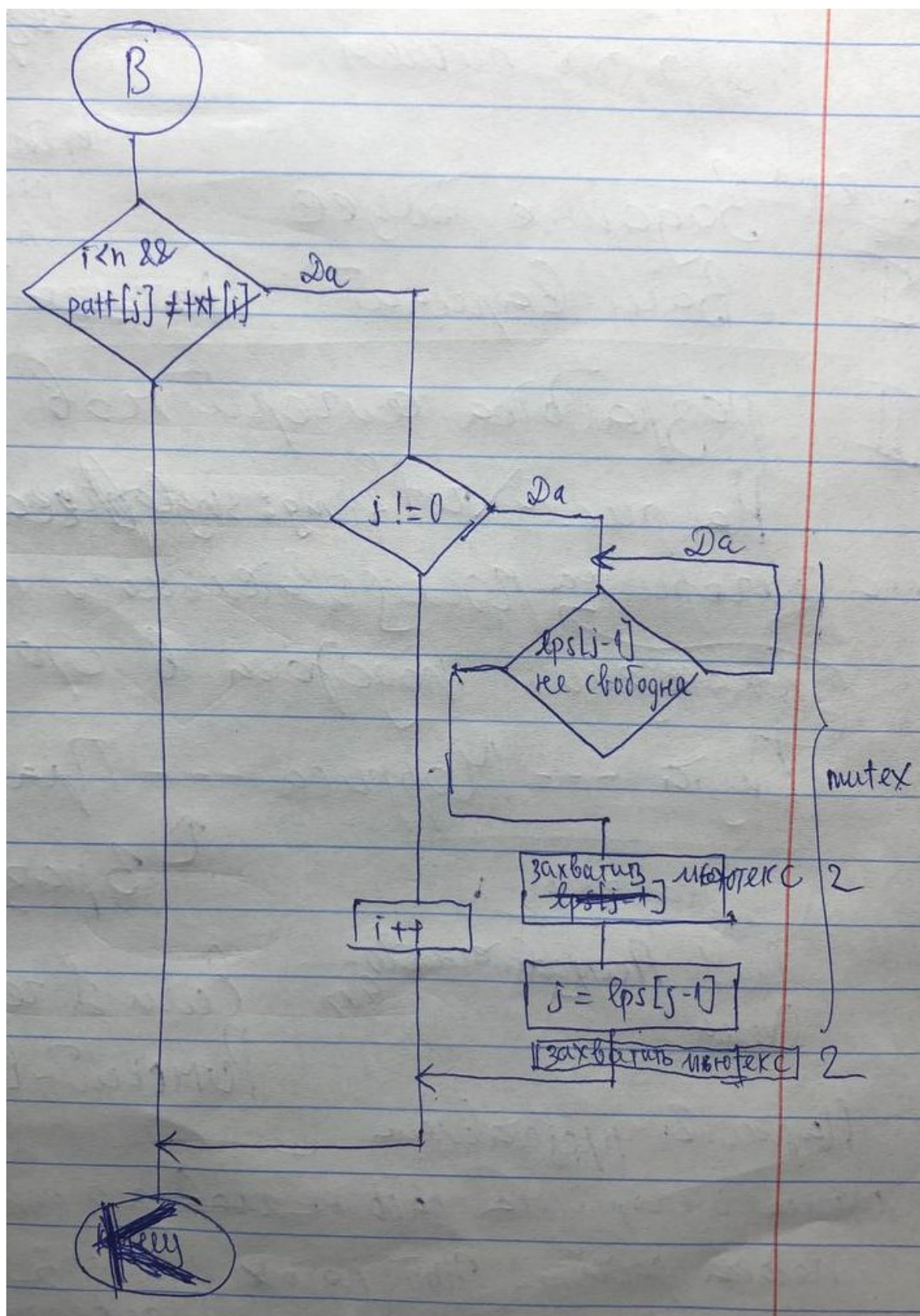


Рисунок 2.8 – Продолжение алгоритма задачи одного потока

3 Технологическая часть

В данном разделе будут указаны средства реализации, будут представлены реализации алгоритмов, а также функциональные тесты.

3.1 Требования к программному обеспечению

Программе, реализующей данные алгоритмы, на вход будут подаваться две строки. Выходным данным такой программы должны быть индексы вхождения подстроки в строке. Программа должна работать в рамках следующих ограничений:

- длины строки и подстроки — положительное целое число;
- длина подстроки не больше длины строки;
- должно быть выдано сообщение об ошибке при вводе некорректно.

3.2 Средства реализации

Реализация данной лабораторной работы выполнялась при помощи языка программирования C++. Данный выбор обусловлен наличием у языка функции *clock()* [4] измерения процессорного времени.

Визуализация графиков с помощью библиотеки *Matplotlib* [5].

3.3 Сведения о модулях программы

Программа состоит из следующих модулей:

- `main.cpp` — точка входа программы;
- `algo.h` — заголовочный файл, содержащий объявления функций, реализующих рассматриваемых алгоритмов;

- algo.cpp — файл, содержащий реализации этих функций;
- measure.h — заголовочный файл, содержащий объявления функций замеров времени работы рассматриваемых алгоритмов;
- measure.cpp — algo файл, содержащий реализации функций замеров времени работы рассматриваемых алгоритмов.

3.4 Реализация алгоритмов

Реализации последовательного и параллельного алгоритмов приведены в листингах 3.1–3.3.

Листинг 3.1 – Последовательный алгоритм Кнута-Морриса-Пратта

```

1 vector<int> algoKMP(string text, string pattern)
2 {
3     int n = text.size(), m = pattern.size();
4     vector<int> lps(m, 0);
5     int i = 0, j = 0;
6     computeLPSArray(pattern, lps);
7     vector<int> res;
8     while ((n - i) >= (m - j))
9     {
10         if (pattern[j] == text[i])
11         {
12             i++;
13             j++;
14         }
15         if (j == m)
16         {
17             res.push_back(i - j);
18             j = lps[j - 1];
19         }
20         else if (i < n && pattern[j] != text[i])
21         {
22             if (j != 0)
23                 j = lps[j - 1];
24             else
25                 i += 1;
26         }
27     }
28     return res;
29 }
```

Листинг 3.2 – Параллельный алгоритм Кнута-Морриса-Пратта

```
1 vector<int> parallelKMP(string text, string pattern, int k)
2 {
3     vector<int> res;
4     int n = text.size(), m = pattern.size();
5     vector<int> lps(m, 0);
6     computeLPSArray(pattern, lps);
7     vector<thread> threads_1;
8     int segmentSize = n / k;
9     int start, end;
10    for (int i = 0; i < k; i++)
11    {
12        start = i * segmentSize;
13        end = (i == k - 1) ? n : (i + 1) * segmentSize;
14        threads_1.emplace_back(kmpSearch, start, end, ref(pattern),
15                                ref(text), ref(res), ref(lps));
16    }
17    for (auto& th : threads_1)
18        th.join();
19    vector<thread> threads_2;
20    for (int i = 1; i <= k - 1; i++)
21    {
22        start = i * segmentSize - (m - 1);
23        end = i * segmentSize + (m - 1);
24        threads_2.emplace_back(kmpSearch, start, end, ref(pattern),
25                                ref(text), ref(res), ref(lps));
26    }
27    for (auto& th : threads_2)
28        th.join();
29    return res;
30 }
```

Листинг 3.3 – Задача одного потока

```
1 void kmpSearch(int start, int end, string& pattern, string& text,
2               vector<int>& res, vector<int> &lps)
3 {
4     int n = end, m = pattern.size();
5     int i = start, j = 0;
6     while ((n - i) >= (m - j))
7     {
8         if (pattern[j] == text[i])
9         {
10            i++;
11            j++;
12        }
13        if (j == m)
```

```

14     {
15         m_res.lock();
16         res.push_back(i - j);
17         m_res.unlock();
18
19         m_lps.lock();
20         j = lps[j - 1];
21         m_lps.unlock();
22     }
23     else if (i < n && pattern[j] != text[i])
24     {
25         if (j != 0)
26         {
27             m_lps.lock();
28             j = lps[j - 1];
29             m_lps.unlock();
30         }
31         else
32             i += 1;
33     }
34 }
35 }

```

3.5 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для функции, реализующей алгоритм поиска подстроки в строке. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

№	Входные данные		Ожидаемый результат
	Строка	Подстрока	Алгоритм КМП
1	Пустая строка	Пустая строка	Сообщение об ошибке
2	a	aa	Сообщение об ошибке
3	abcdabc	bc	[1, 5]
4	daymldayaaaymtstadayamkl	day	[0, 5, 17]

Вывод

Были представлены листинги реализаций двух версий алгоритма КМП — последовательного и параллельного. Также в данном разделе была приведены функциональные тесты.

4 Исследовательская часть

В данном разделе будут приведены демонстрации работы программы, и будет проведен сравнительный анализ реализованного алгоритма Кнута-Морриса-Пратта по времени.

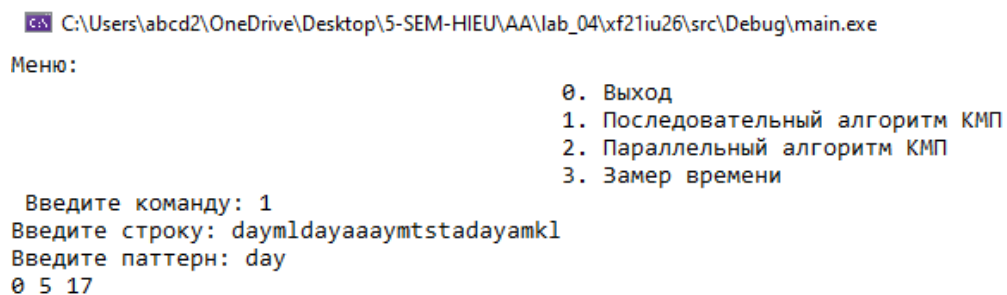
4.1 Технические характеристики

Тестирование проводилось на устройстве со следующими техническими характеристиками:

- операционная система Window 10 Home Single Language;
- память 8 Гб;
- процессор 11th Gen Intel(R) Core(TM) i7-1165G7 2.80 ГГц, 4 ядра.

4.2 Демонстрация работы программы

На рисунках 4.1 и 4.2 представлен результат работы программы. В каждом примере пользователем введены строка и подстрока и получены все индексы вхождения.



```
C:\Users\abcd2\OneDrive\Desktop\5-SEM-HIEU\AA\lab_04\xf21iu26\src\Debug\main.exe
Меню:
0. Выход
1. Последовательный алгоритм КМП
2. Параллельный алгоритм КМП
3. Замер времени
Введите команду: 1
Введите строку: daymldayaaaumtstadayamk1
Введите паттерн: day
0 5 17
```

Рисунок 4.1 – Демонстрация работы программы при последовательном алгоритме

```

Меню:
0. Выход
1. Последовательный алгоритм КМП
2. Параллельный алгоритм КМП
3. Замер времени

Введите команду: 2
Введите строку: daymldayaaaumtstadayamkl
Введите паттерн: day
Введите количество потоков: 4
0 5 17

```

Рисунок 4.2 – Демонстрация работы программы при параллельном алгоритме

4.3 Время выполнения реализаций алгоритмов

Результаты замеров времени работы реализаций алгоритмов поиска подстроки в строке преведены в таблице 4.1. Для параллельной реализации алгоритма выбрано количество потоков, равное 4, поскольку именно столько логических ядер имеет процессор ноутбука.

Таблица 4.1 – Результаты замеров времени

Размер строки	Размер подстроки	Без многопоточности	4 потока
100000	5	23.4	17.2
200000	5	36.5	26.4
300000	5	55.4	36.1
400000	5	72.8	45
500000	5	92.7	54

По таблице 4.1 был построен график, который иллюстрирует зависимость времени, затраченного реализациями алгоритмов КМП, от размера строки — рис. 4.3.

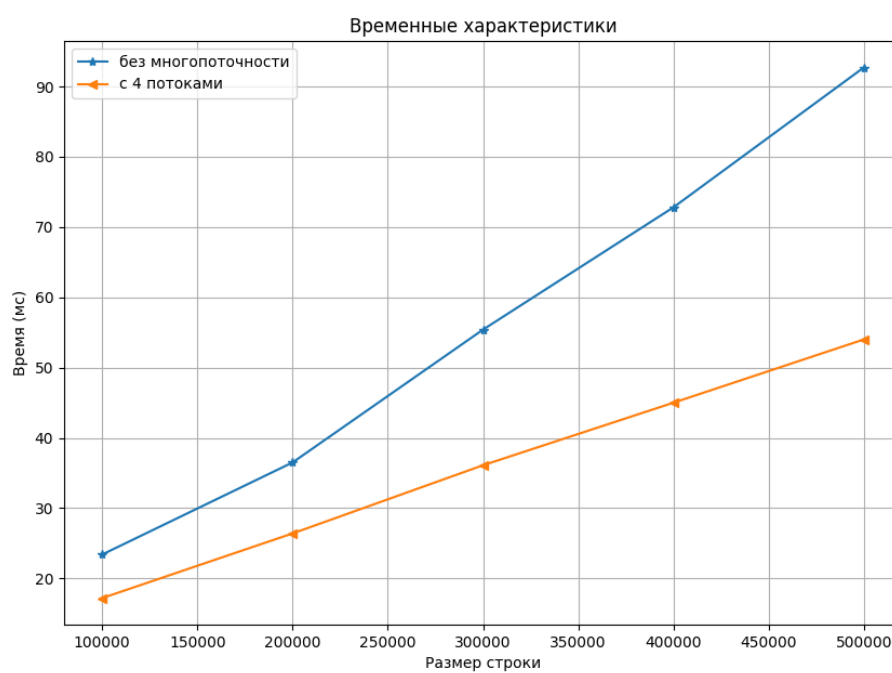


Рисунок 4.3 – Сравнение времени работы алгоритма без распараллеливания и с 4 вспомогательными потоками при разных размерах строки

Исходя из этих данных можно понять, что параллельный алгоритм с использованием 4 потоками работает быстрее последовательного алгоритма КМП.

В таблице 4.2 приведены результаты замеров по времени параллельного алгоритма КМП при разном количество потоков.

Таблица 4.2 – Результаты замеров времени

Количество потоков	Время выполнения
1	962.2
2	611
4	488.1
8	504.8
16	506.3

По таблице 4.2 был построен график, который иллюстрирует зависимость времени, затраченного реализацией параллельного алгоритма КМП, от количества потоков — рис. 4.4.

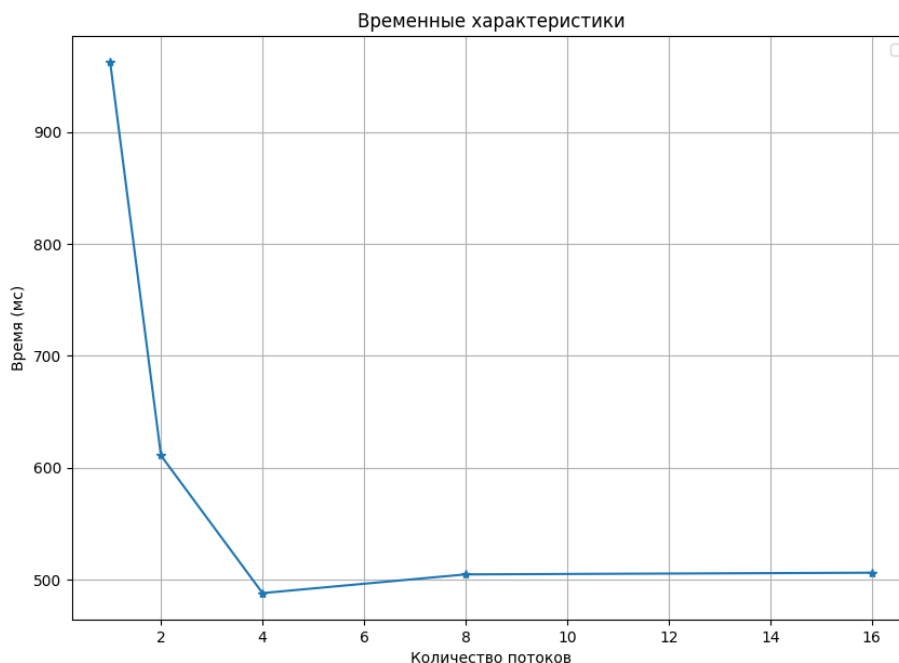


Рисунок 4.4 – Сравнение времени работы алгоритма с распараллеливанием на различное количество потоков при размере строки 5000000

Исходя из этих данных можно понять, параллельный алгоритм

работает быстрее всего с 4 потоками.

4.4 Вывод

По графикам видно, что при использовании 4 вспомогательных потоков, многопоточная реализация алгоритма значительно эффективнее по времени реализации без многопоточности при работе с строкой размером 5000000. Данное количество потоков обусловлено тем, что на ноутбуке, на котором проводились замеры времени, имеется всего 4 логических ядра, а следовательно, количество потоков, при котором потоки будут распределены между всеми ядрами равномерно, равно 4. Именно поэтому лучшие результаты достигаются именно на 4 потоках. Исходя из построенных графиков, можно сделать вывод, что распараллеливание кода значительно увеличивает эффективность алгоритма поиска подстроки в строке по времени.

Заключение

При использовании 4 вспомогательных потоков, многопоточная реализация алгоритма значительно эффективнее по времени реализации без многопоточности при работе с строкой размером 5000000. Данное количество потоков обусловлено тем, что на ноутбуке, на котором проводились замеры времени, имеется всего 4 логических ядра, а следовательно, количество потоков, при котором потоки будут распределены между всеми ядрами равномерно, равно 4. Именно поэтому лучшие результаты достигаются именно на 4 потоках. Исходя из построенных графиков, можно сделать вывод, что распараллеливание кода значительно увеличивает эффективность алгоритма поиска подстроки в строке по времени.

Цель лабораторной работы была достигнута, в ходе выполнения лабораторной работы были решены все задачи:

- описаны последовательный и параллельный алгоритмы поиска подстроки в строке Кнута-Морриса-Пратта;
- построены схемы данных алгоритмов;
- создано программное обеспечение, реализующее рассматриваемые алгоритмы;
- проведен сравнительный анализ по времени для реализованного алгоритма;
- подготовлен отчет о выполненной лабораторной работе.

Список использованных источников

1. Многопоточность в C++. Основные понятия [Электронный ресурс]. Режим доступа: <https://radioprogram.ru/post/1402>
2. Алгоритм Кнута, Мориса и Пратта [Электронный ресурс]. Режим доступа: http://avanta.vvsu.ru/met_supply/312/algorithm_knuta.htm
3. Распараллеливание алгоритма сопоставления строк КМР [Электронный ресурс]. Режим доступа: https://www.researchgate.net/publication/269897379_Parallelization_of_KMP_String_Matching_Algorithm_on_Different_SIMD_Architectures_Multi-Core_and_GPGPU's
4. C library function clock() [Электронный ресурс]. Режим доступа: <https://en.cppreference.com/w/cpp/chrono/c/clock>
5. Matplotlib documentation [Электронный ресурс]. Режим доступа: <https://matplotlib.org/stable/index.html>