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 Кузнецова А. В.

Группа ИУ7-51Б

Оценка (баллы)

Преподаватель Волкова Л. Л.

Москва — 2022 г.

Оглавление

Введение

Одной из задач программирования является ускорение решения вычислительных задач. Один из способов ее решения — использование параллельных вычислений.

В компьютерной архитектуре многопоточность — способность центрального процессора (CPU) или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой. Этот подход отличается от многопроцессорности, так как многопоточность процессов и потоков совместно использует ресурсы одного или нескольких ядер: вычислительных блоков, кэш-памяти ЦПУ или буфера перевода с преобразованием (TLB) [1].

Целью данной лабораторной работы является получение навыка организации параллельных вычислений на основе нативных потоков на примере обработки разреженных матриц, представленных с помощью кольцевой KPM-схемы. Для достижения поставленной цели требуется решить задачи, представленные ниже.

1. Описать KPM-схему хранения матриц.
2. Разработать и реализовать однопоточный алгоритм преобразования верхнетреугольной матрицы к набору матриц, в каждой из которых исключены элементы, большие k, где k от 1 до Q (Q-вход).
3. Разработать и реализовать многопоточную версию данного алгоритма.
4. Создать схемы изучаемых алгоритмов.
5. Определить средства программной реализации.
6. Выполнить замеры процессорного времени работы реализаций алгоритма.
7. Провести сравнительный анализ по времени работы реализаций алгоритмов.
8. Подготовить отчет о выполненной лабораторной работе.

1 Аналитическая часть

В данном разделе представлено теоретическое описание КРМ-схемы хранения разреженных матриц и заданного алгоритма.

1.1 КРМ-схема хранения разреженных матриц

Для начала следует описать схему хранения разреженной матрицы, предложенную Кнутом. Ненулевые элементы хранятся в компактной форме в одномерном массиве AN. Информация о положении ненулевых элементов в матрице хранится двумя дополнительными параллельными одномерными массивами — I и J; здесь для каждого ненулевого элемента содержатся его строчный и столбцовый индексы. Итак, для каждого $a_{ij} \neq 0$ в памяти находится тройка (a_{ij}, i, j). Далее, чтобы можно было легко отыскивать элементы произвольной строки или столбца матрицы, необходимы еще пара указателей для каждой тройки, а также указатели входа для строк и столбцов, сообщающие начало каждого строчного или столбцового списка. Пусть NR («next nonzero element in the same row» — «следующий ненулевой элемент той же строки») — массив, хранящий строчные указатели, а NC («next nonzero element in the same column» — «следующий ненулевой элемент того же столбца») — массив столбцовых указателей. Пять массивов AN, I, J, NR и NC имеют одинаковую длину, и их одноименные позиции соответствуют друг другу. Пусть JR и JC — массивы, содержащие указатели входа для строк и столбцов, расположенные в соответствии с порядком строк и столбцов матрицы. Тогда рассмотрим матрицу на рис. 1.1, её представление с помощью схемы Кнута — на рис. 1.2.

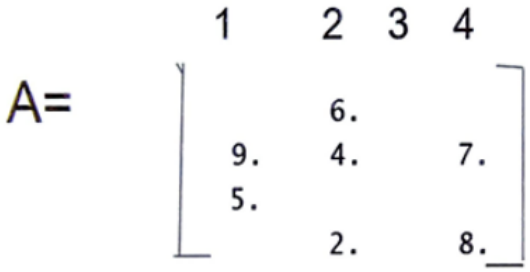

Рисунок 1.1 – Разреженная матрица

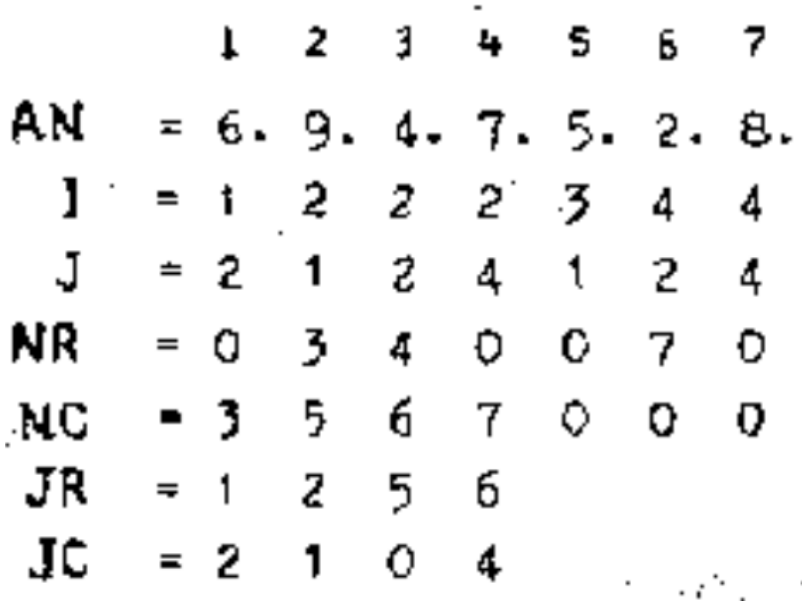

Рисунок 1.2 – Схема Кнута

Рейнболдт и Местеньи предложили модификацию схемы Кнута, сохраняющую ее ценные свойства, но использующую значительно меньше накладных расходов по памяти. Она получила название схемы Кнута-Рейнболдта-Местеньи, или кольцевая КРМ-схема. Связные списки строк и столбцов закольцовываются, а начальные позиции списков включаются в указатели входа. Списки, ассоциированные со строками (столбцами), попарно не пересекаются и потому могут быть совместно хранимы одним массивом NR (для столбцов — NC). Для матрицы на рис. 1.1 [2] приведено ее представление с помощью КРМ-схемы на рис. 1.3. Эта схема более плотная по сравнению со схемой Кнута. Однако, если приходится просматривать элементы некоторой строки (или столбца), то в сжатом формате нет никакой информации о столбцовых (строчных) индексах этих элементов [3].

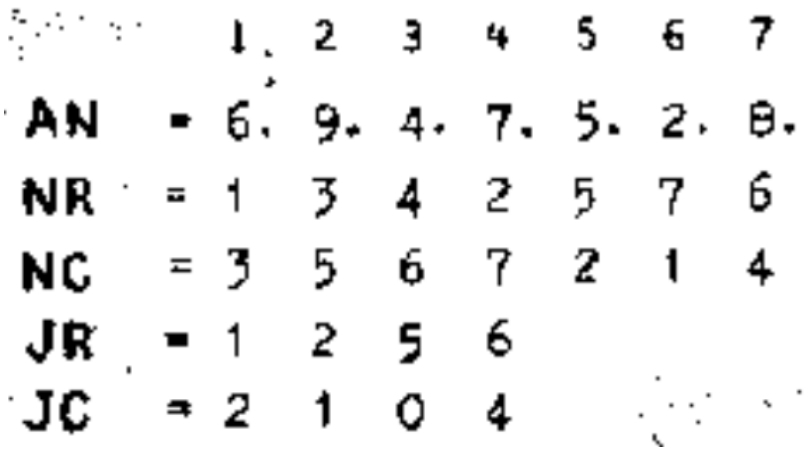

Рисунок 1.3 – Кольцевая КРМ-схема

1.2 Описание алгоритма

Для получения из верхнетреугольной матрицы набора матриц, из которых исключены значения, меньшие k, где $k \in [1, Q]$ (Q является входом алгоритма), необходимо в цикле по k, $k \in [1, Q]$, на каждой итерации удалить элементы, большие k. Для удаления элемента из матрицы нужно удалить его из массива AN, хранящего значения ненулевых элементов. Затем в массивах NR и NC изменить следующее: тот, кто ссылался на удаляемый элемент, теперь должен ссылаться на элемент, на который ссылался удаляемый. Значения номеров элементов, большие удаляемого, в массивах NR и NC должны уменьшиться на 1. В массивах JR и JC в случае, когда удаляется последний элемент в строке (столбце) записать 0 в позицию массива JR (JC), равную номеру строки. В остальных случаях, когда удаляемый элемент является первым в строке (столбце) и в этой строке (столбце) есть другие элементы, надо вместо номера удаляемого элемента в массив JR (JC) записать номер элемента, на который ссылается удаляемый элемент в массиве NR (NC). Значения номеров элементов, большие удаляемого, в массивах JR и JC также должны уменьшиться на 1.

Вывод

В данном разделе были описаны основные положения КРМ-схемы хранения разреженных матриц и заданного вариантом алгоритма обработки упакованной разреженной матрицы.

2 Конструкторская часть

В данном разделе будет рассмотрена схема вышеизложенного алгоритма.

2.1 Разработка алгоритмов

На рисунках 2.1 – 2.2 представлен алгоритм удаления элементов, больших k, где $k \in [1, Q]$ (Q — вход).

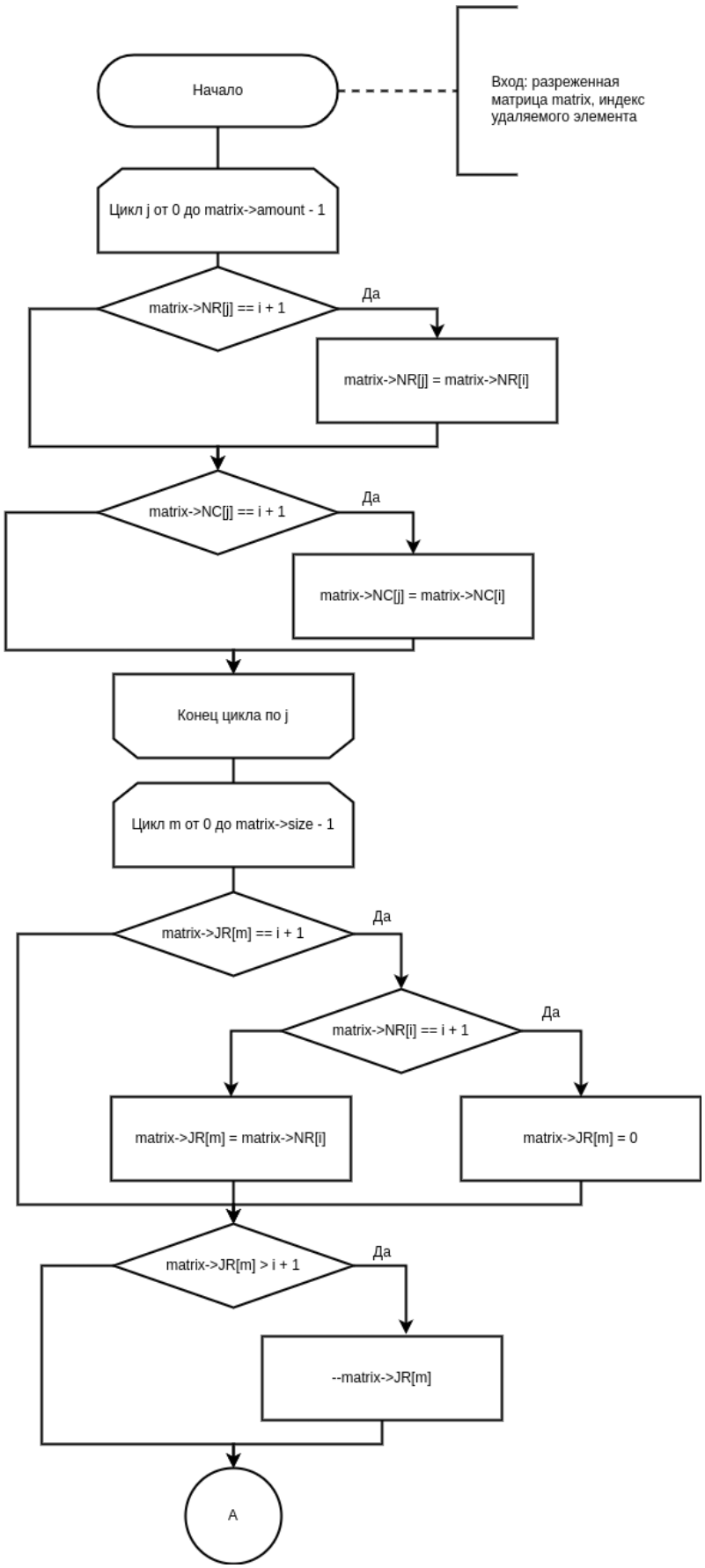

Рисунок 2.1 – Схема алгоритма удаления элемента (1 часть)

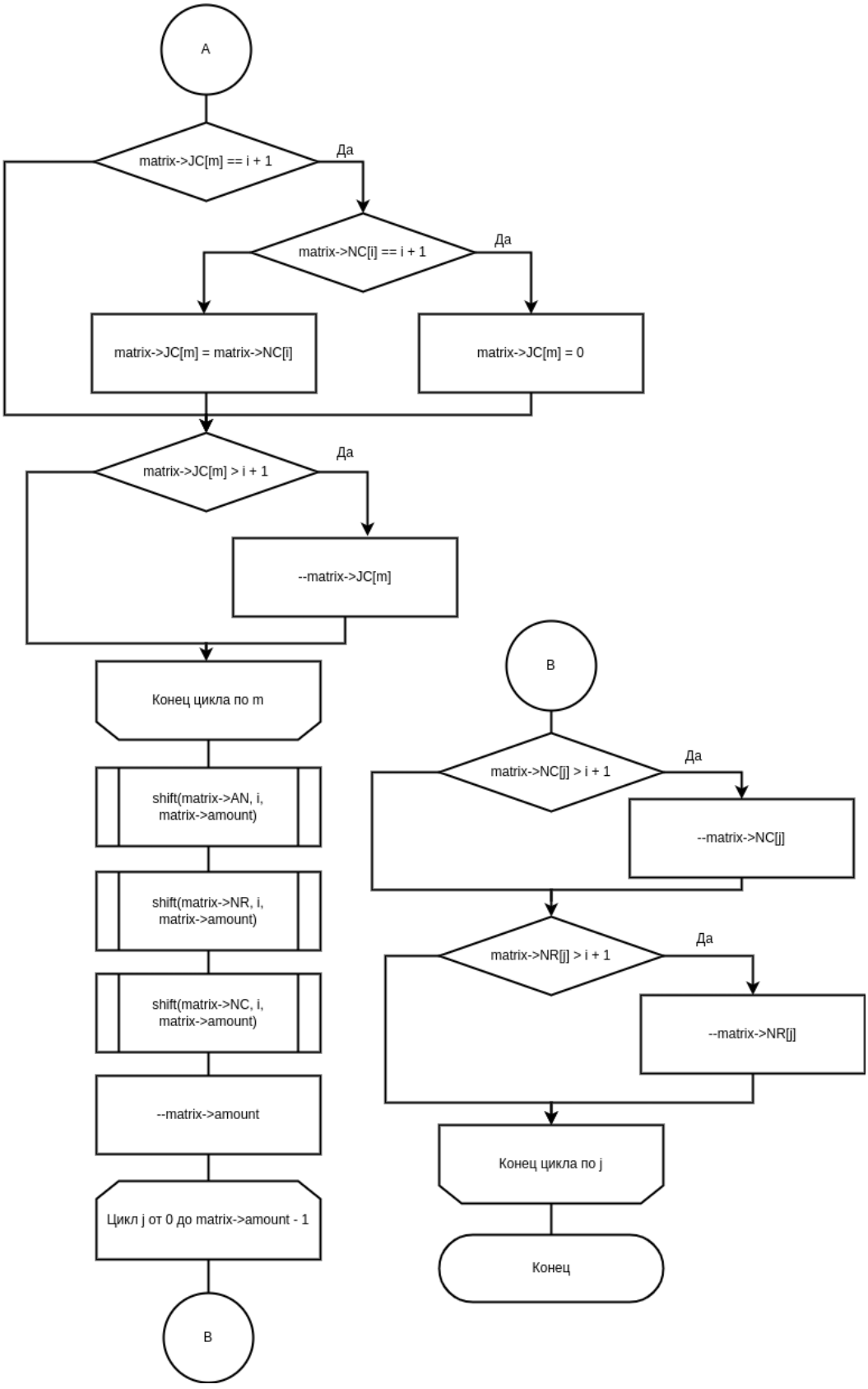

Рисунок 2.2 – Схема алгоритма удаления элемента (2 часть)

На рисунке 2.3 представлен последовательный алгоритм получения из верхнетреугольной матрицы набора матриц, из которых исключены значе-

ния, большие k, где $k \in [1, Q]$ (Q — вход).

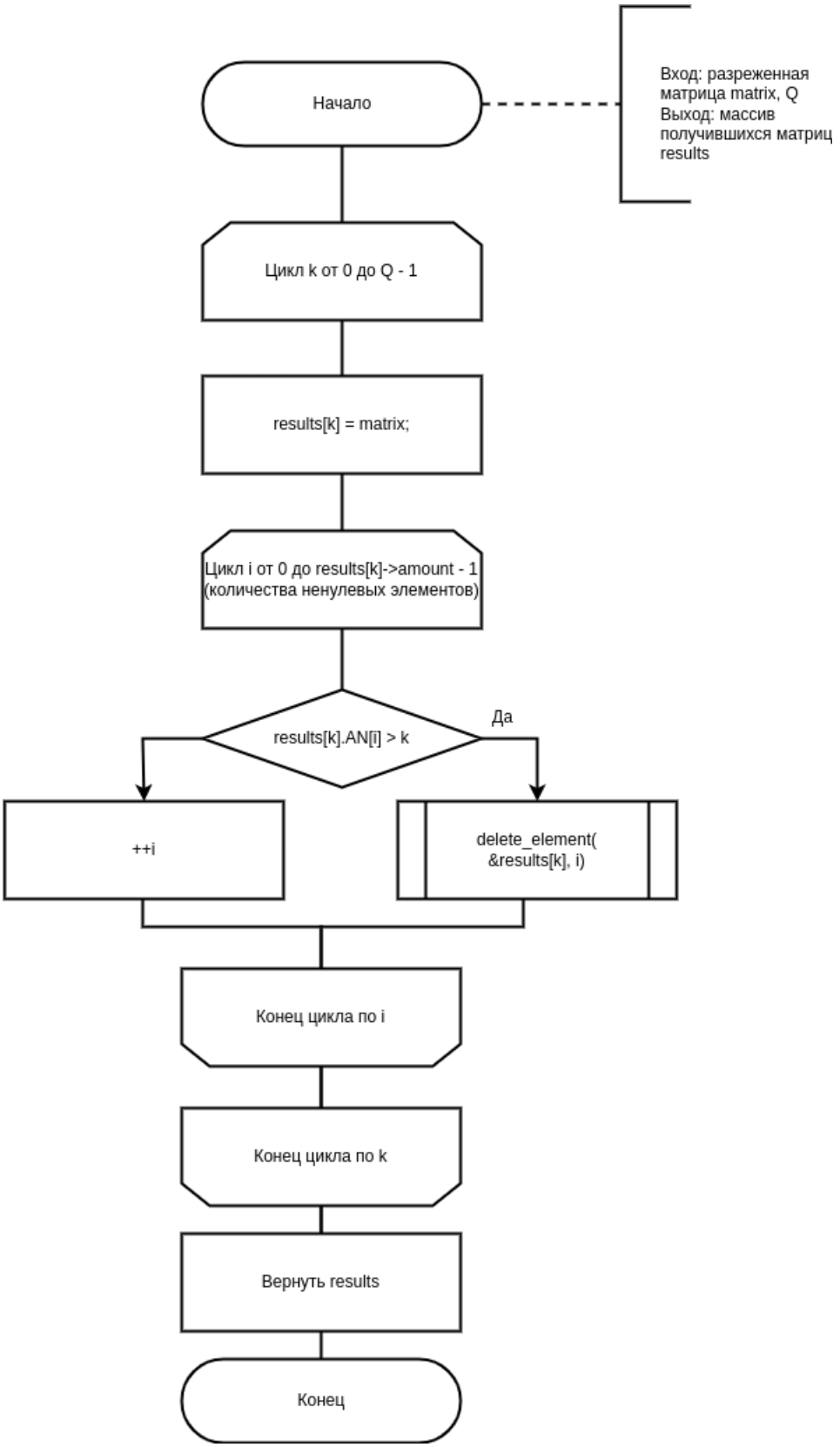

Рисунок 2.3 – Схема последовательного алгоритма

На рисунке 2.4 представлен алгоритм запуска и ожидания потоков.

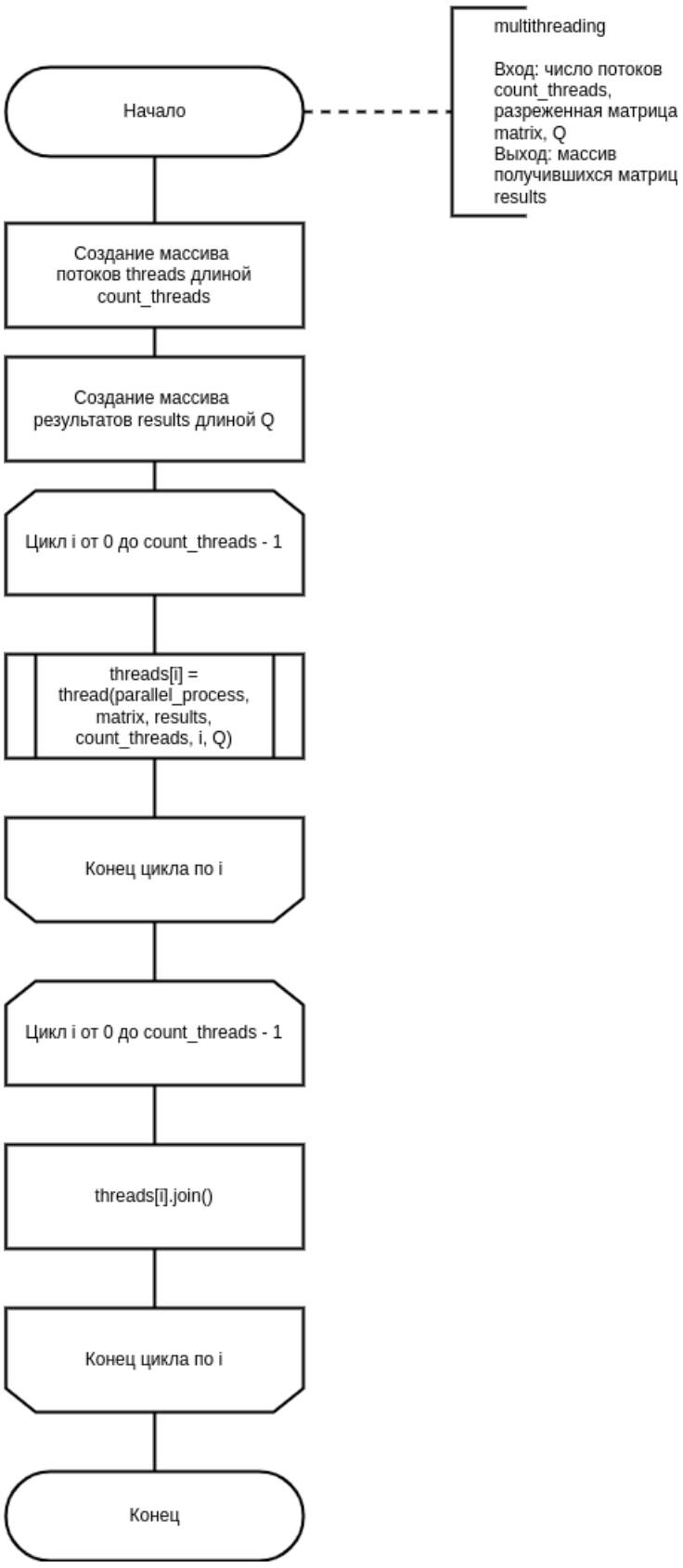

Рисунок 2.4 – Схема запуска и одидания потоков

На рисунке 2.5 представлен параллельный алгоритм получения из верхнетреугольной матрицы набора матриц, из которых исключены значения,

большие k, где $k \in [1, Q]$ (Q — вход).

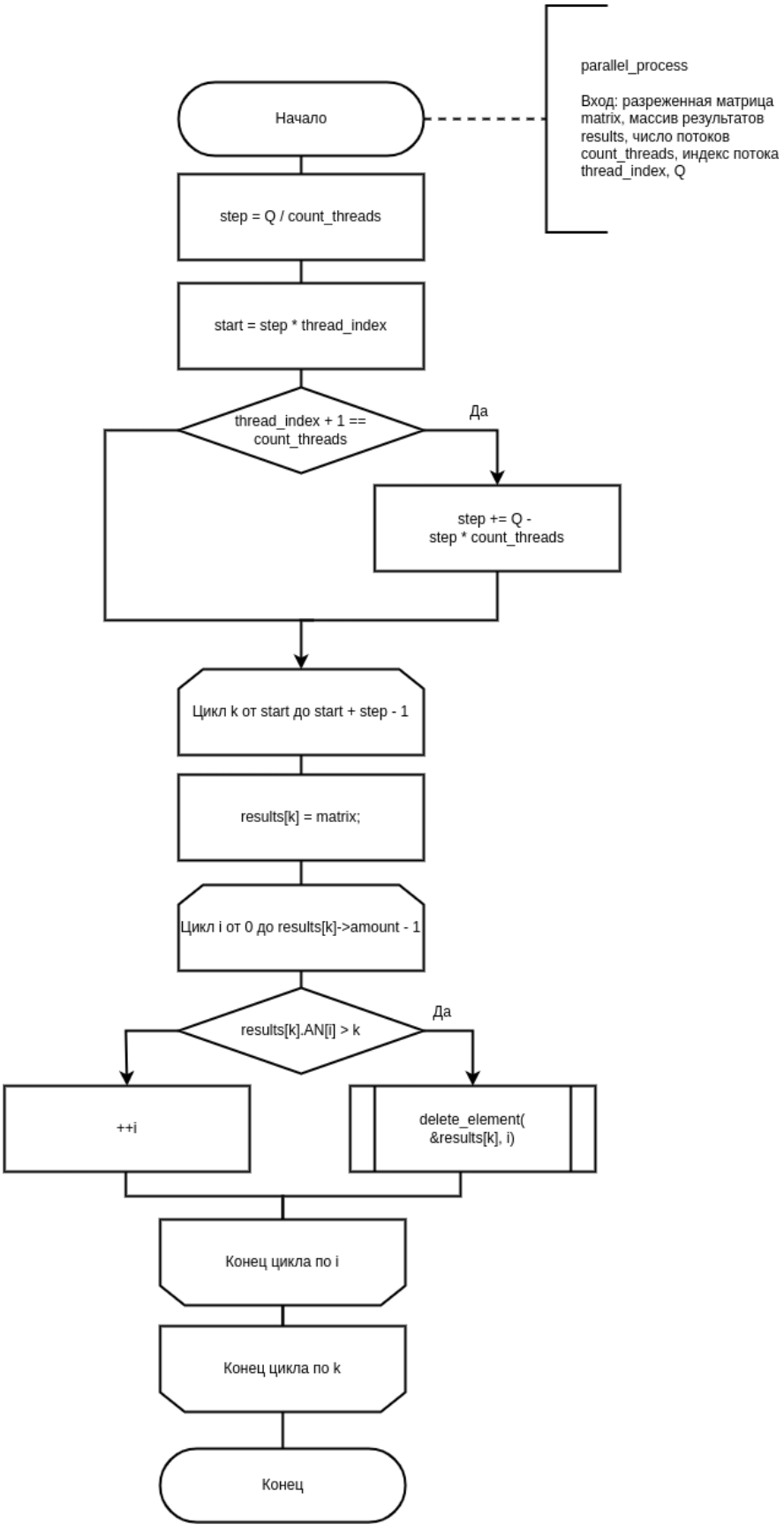

Рисунок 2.5 – Схема параллельного алгоритма

Вывод

На основе теоретических данных, полученных из аналитического раздела, была построена схема требуемого алгоритма.

3 Технологическая часть

В данном разделе приведены средства реализации, сведения о модулях программы, листинги кода, тесты.

3.1 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык $C++$ [4]. Данный выбор обусловлен наличием инструментов для реализации принципов многопоточного программирования.

Замеры времени проводились при помощи функции $std :: chrono :: system\_clock :: now(...)$ из библиотеки $chrono$ [5].

Реализация графического представления замеров времени производилась при помощи языка программирования $Python$ [6], так как данный язык программирования представляет графическую библиотеку для визуализации данных.

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

— main.cpp — файл, содержащий точку входа в программу, в нем происходит вызов алгоритмов;

— matrix.cpp и matrix.h — файлы, содержащие алгоритмы работы с матрицами;

— constants.h — файл, содержащий необходимые константы и объявления;

— measurements.cpp и measurements.h — файл, содержащий функции замеров времени работы алгоритмов;

3.3 Реализация алгоритмов

В листингах 3.1–3.3 представлены реализации последовательного и параллельного алгоритмов обработки упакованной разреженной матрицы.

Листинг 3.1 – Реализация последовательного алгоритма обработки упакованной разреженной матрицы при фиксированном Q

```
1
2         void delete_elements_from_matrix(matrix_t *matrix, int k)
             {
3             for (int i = 0; i < matrix->amount;) {
4                 if (matrix->AN[i] > k) {
5                     for (int j = 0; j < matrix->amount; ++j) {
6                         if (matrix->NR[j] == i + 1) {
7                             matrix->NR[j] = matrix->NR[i];
8                         }
9                         if (matrix->NC[j] == i + 1) {
10                            matrix->NC[j] = matrix->NC[i];
11                        }
12                    }
13                    for (int m = 0; m < matrix->size; ++m) {
14                        if (matrix->JR[m] == i + 1) {
15                            if (matrix->NR[i] == i + 1) {
16                                matrix->JR[m] = 0;
17                            } else {
18                                matrix->JR[m] = matrix->NR[i];
19                            }
20                        }
21                        if (matrix->JR[m] > i + 1) {
22                            --matrix->JR[m];
23                        }
24                        if (matrix->JC[m] == i + 1) {
25                            if (matrix->NC[i] == i + 1) {
26                                matrix->JC[m] = 0;
27                            } else {
28                                matrix->JC[m] = matrix->NC[i];
29                            }
30                        }
31                        if (matrix->JC[m] > i + 1) {
32                            --matrix->JC[m];
33                        }
34                    }
35                    delete_element(matrix->NR, i, matrix->amount);
36                    delete_element(matrix->AN, i, matrix->amount);
37                    delete_element(matrix->NC, i, matrix->amount);
38                    --matrix->amount;
```

```
39                      for (int j = 0; j < matrix->amount; ++j) {
40                          if (matrix->NC[j] > i + 1) {
41                              --matrix->NC[j];
42                          }
43                          if (matrix->NR[j] > i + 1) {
44                              --matrix->NR[j];
45                          }
46                      }
47                  } else {
48                      ++i;
49                  }
50              }
51          }
```

Листинг 3.2 – Однопоточная реализация алгоритма обработки упакованной разреженной матрицы

```
1
2  matrix_t *process (matrix_t matrix, int Q) {
3      cout << "RESULT:" << endl;
4      matrix_t *results = new matrix_t[Q];
5      for (int k = 0; k < Q; ++k) {
6          results[k].amount = matrix.amount;
7          results[k].size = matrix.size;
8          results[k].JR = new int[matrix.size];
9          results[k].JC = new int[matrix.size];
10         results[k].AN = new int[matrix.amount];
11         results[k].NR = new int[matrix.amount];
12         results[k].NC = new int[matrix.amount];
13
14         memcpy(results[k].JR, matrix.JR, matrix.size *
               sizeof(int));
15         memcpy(results[k].JC, matrix.JC, matrix.size *
               sizeof(int));
16         memcpy(results[k].AN, matrix.AN, matrix.amount *
               sizeof(int));
17         memcpy(results[k].NR, matrix.NR, matrix.amount *
               sizeof(int));
18         memcpy(results[k].NC, matrix.NC, matrix.amount *
               sizeof(int));
19
20         delete_elements_from_matrix(&results[k], k + 1);
```

```
21      }
22      return results;
23  }
```

Листинг 3.3 – Многопоточная реализация алгоритма обработки упакованной разреженной матрицы

```
1
2   void parallel_process(matrix_t* matrix, std::vector<matrix_t>
        results, int count_threads, int thread_index, int Q)
3   {
4       cout << endl << "==========␣THREAD␣" << thread_index + 1 << "␣"
            << "START" << endl;
5
6       int step = Q / count_threads;
7       int start = thread_index * step;
8
9       if (thread_index + 1 == count_threads)
10      {
11          step += Q - step * count_threads;
12      }
13
14      for (int k = start; k < start + step; k++)
15      {
16          results[k].amount = matrix->amount;
17          results[k].size = matrix->size;
18          results[k].JR = new int[matrix->size];
19          results[k].JC = new int[matrix->size];
20          results[k].AN = new int[matrix->amount];
21          results[k].NR = new int[matrix->amount];
22          results[k].NC = new int[matrix->amount];
23
24          memcpy(results[k].JR, matrix->JR, matrix->size *
                sizeof(int));
25          memcpy(results[k].JC, matrix->JC, matrix->size *
                sizeof(int));
26          memcpy(results[k].AN, matrix->AN, matrix->amount *
                sizeof(int));
27          memcpy(results[k].NR, matrix->NR, matrix->amount *
                sizeof(int));
28          memcpy(results[k].NC, matrix->NC, matrix->amount *
                sizeof(int));
```

```
29          delete_elements_from_matrix(&results[k], k + 1);
30          cout << "k = " << k + 1 << endl;
31          print_matrix(&results[k]);
32      }
33
34      cout << endl << "========= THREAD " << thread_index + 1 << " "
           << "END" << endl;
35  }
36
37  std::vector<matrix_t> multithreading(int count_threads, matrix_t*
        matrix, int Q)
38  {
39      std::vector<std::thread> threads(count_threads);
40      std::vector<matrix_t> results(Q);
41
42      for (int i = 0; i < count_threads; i++)
43      {
44          threads[i] = std::thread(parallel_process,
               std::ref(matrix), std::ref(results), count_threads, i,
               Q);
45      }
46
47      for (int i = 0; i < count_threads; i++)
48      {
49          threads[i].join();
50      }
51
52      return results;
53  }
```

3.4 Тестирование

В таблице 3.1 приведены функциональные тесты для реализованного алгоритма.

При проведении функционального тестирования, полученные результаты работы программы совпали с ожидаемыми. Таким образом, функциональное тестирование пройдено успешно.

Таблица 3.1 – Функциональные тесты

Входная матрица и Q	Ожидаемый результат
[]	[]
AN:8, 7, 4, 6, 2, 9 NR:2, 3, 1, 5, 4, 6 NC:1, 4, 5, 2, 6, 3 JR:1, 4, 6 JC:1, 2, 3 Q = 2	k = 1 AN: NR: NC: JR:0, 0, 0 JC:0, 0, 0 k = 2 AN:2 NR:1 NC:1 JR:0, 1, 0 JC:0, 0, 1

Вывод

Были реализованы однопоточная и многопоточная версии заданного алгоритма.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программы, и будет проведен сравнительный анализ реализаций алгоритма по затраченному процессорному времени.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование, приведены ниже:

— операционная система Manjaro Linux [7];

— память 7,6 ГБ;

— процессор 8 × Intel® Core™ i5-10210U CPU @ 1.60 ГГц [8] с 4 физическими ядрами и 8 логическими.

4.2 Демонстрация работы программы

На рисунке 4.1 приведен пример работы программы. На этом рисунке пользователь выбирает из меню пункт 1 — однопоточную реализацию алгоритма, вводит Q и размерность матрицы и получает на выходе набор матриц, в каждой из которых значения элементов <= k, где $k \in [1, Q]$.

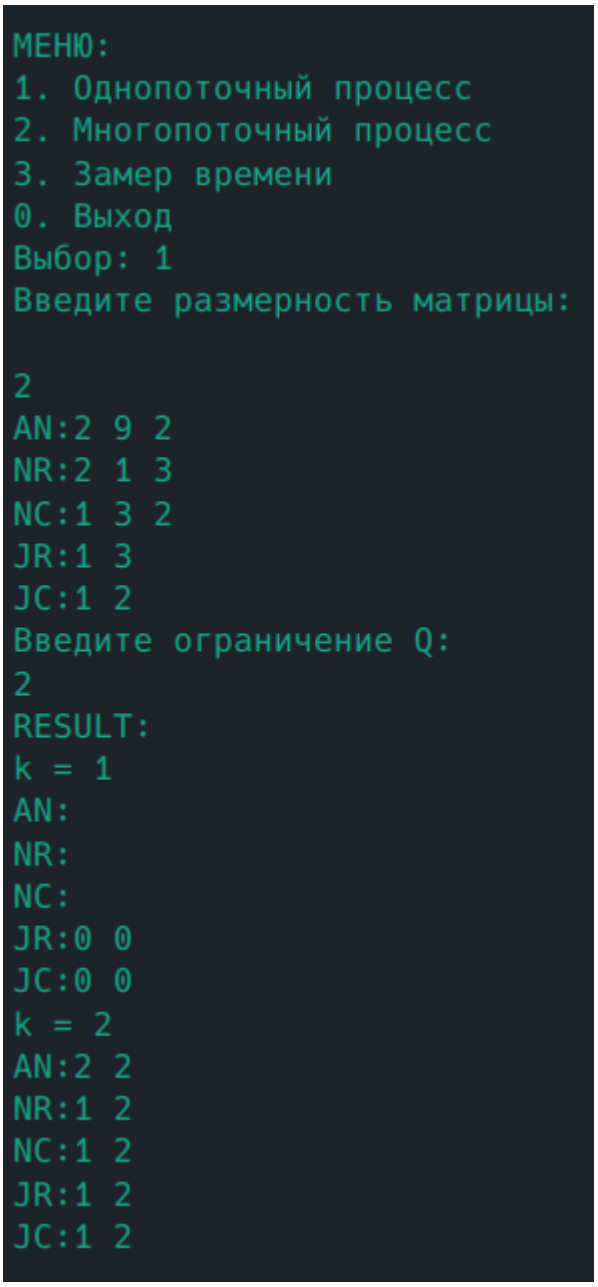

Рисунок 4.1 – Пример работы программы

4.3 Временные характеристики

Функция $std :: chrono :: system\_clock :: now(...)$ из библиотеки $chrono$ языка программирования $C++$ возвращает процессорное время в секундах — значение типа float.

На рисунке 4.2 приведены результаты замеров времени выполнения однопоточной и многопоточной реализаций заданного алгоритма.

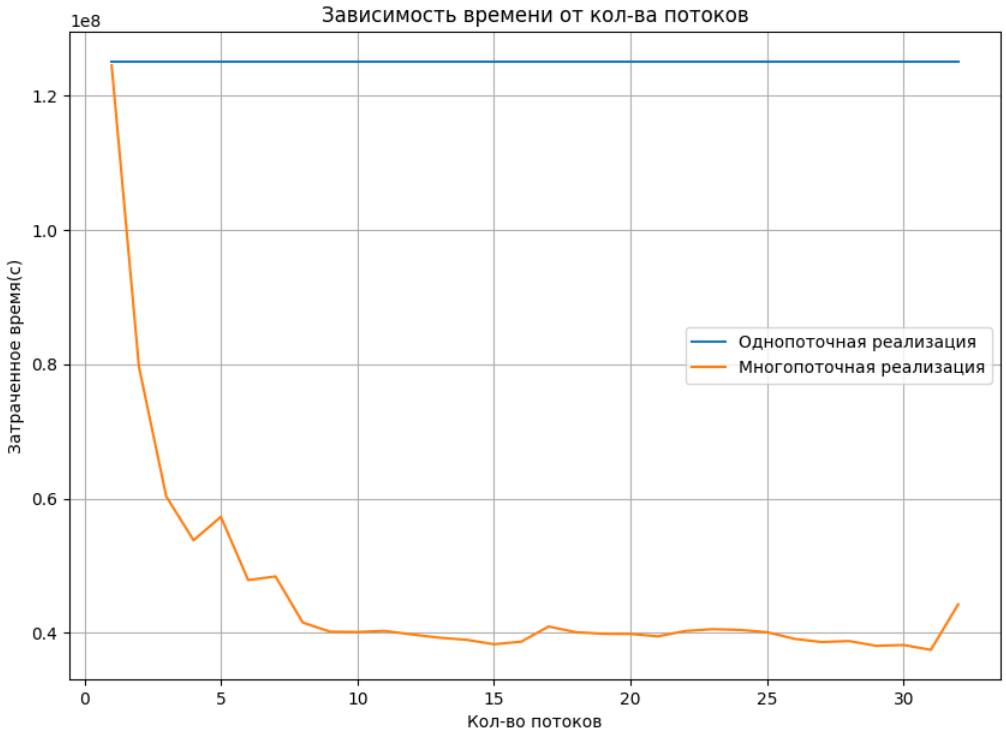

Рисунок 4.2 – Сравнение времени выполнения реализаций алгоритмов

4.4 Вывод

Приведенные характеристики времени показывают, что многопоточная реализация выигрывает по времени у последовательной реализации, достигается ускорение в 3 раза.

Заключение

В результате выполнения данной лабораторной работы были рассмотрены однопоточная и многопоточная версии заданного алгоритма обработки разреженной матрицы, упакованной по КРМ-схеме, построены схемы, соответствующие данным алгоритмам, и выполнена их реализация. Можно сделать вывод, что для больших значений Q стоит применять многопоточную реализацию заданного алгоритма. Проведенные замеры времени работы реализаций показали, что на выбранной архитектуре ЭВМ достигается ускорение в 3 раза при числе потоков, кратным числу логических ядер процессора.

В рамках выполнения работы цель достигнута: получен навык организации параллельных вычислений на основе нативных потоков на примере обработки разреженных матриц, представленных с помощью кольцевой КРМ-схемы.

Решены все задачи:

— описана КРМ-схема хранения матриц;

— разработан и реализован однопоточный алгоритм преобразования верхнетреугольной матрицы к набору матриц, в каждой из которых исключены элементы, большие k, где k от 1 до Q (Q-вход);

— разработана и реализована многопоточная версия данного алгоритма;

— созданы схемы изучаемых алгоритмов;

— определены средства программной реализации;

— выполнены замеры процессорного времени работы реализаций алгоритма;

— проведен сравнительный анализ по времени работы реализаций алгоритмов;

— подготовлен отчет о выполненной лабораторной работе.

Список используемых источников

[1] Многопоточность [Электронный ресурс]. Режим доступа: https://ru.bmstu.wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D0%BE%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C (дата обращения: 25.11.2022).

[2] А. В. Силантьева. Лекции по типам и структурам данных. — М.: МГТУ им. Н.Э.Баумана. — 419 с.

[3] С. Писсанецки. Технология разреженных матриц. — М.: Мир, 1988. — 410 с.

[4] Документация по языку C++ [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/cpp/cpp/?view=msvc-170 (дата обращения: 25.11.2022).

[5] Date and time utilities [Электронный ресурс]. Режим доступа: https://en.cppreference.com/w/cpp/chrono (дата обращения: 25.11.2022).

[6] Welcome to Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 25.11.2022).

[7] Manjaro Linux [Электронный ресурс]. Режим доступа: https://manjaro.org (дата обращения: 25.11.2022).

[8] Процессор Intel® Core™ i5-10210U [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/195436/intel-core-i510210u-processor-6m-cache-up-to-4-20-ghz.html (дата обращения: 25.11.2022).