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 №5

по курсу «Анализ Алгоритмов»

на тему: «Организация асинхронного взаимодействия потоков вычисления на примере конвейерных вычислений»

Студент ИУ7-51Б (Группа) _____ (Подпись, дата) Савинова М. Г. (Фамилия И. О.)

Преподаватель _____ (Подпись, дата) Волкова Л. Л. (Фамилия И. О.)

Преподаватель _____ (Подпись, дата) Строганов Ю.В. (Фамилия И. О.)

2024 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной лабораторной работы является описание параллельных конвейерных вычислений.

Для достижения поставленной цели необходимо решить следующие задачи:

1) описать организацию конвейерной обработки данных;

2) описать алгоритмы обработки данных, которые будут использоваться в текущей лабораторной работе;

3) определить средства программной реализации;

4) реализовать программу, выполняющую конвейерную обработку с количеством лент не менее трех в однопоточной и многопоточной реализаций;

5) сравнить и проанализировать реализации алгоритмов по затраченному времени.

1 Аналитический раздел

В данном разделе приведена информация, касающаяся основ конвейерной обработки данных.

1.1 Конвейерная обработка данных

Конвейер — организация вычислений, при которой увеличивается количество выполняемых инструкций за единицу времени за счет использования принципов параллельности.

Конвейеризация в компьютерной обработке данных основана на разбиении выполнения функций на более мелкие этапы, называемые ступенями, и выделении отдельной аппаратуры для каждой из них. Это позволяет организовать передачу данных от одного этапа к следующему, что увеличивает производительность за счет одновременного выполнения нескольких команд на различных ступенях конвейера.

Хотя конвейеризация не уменьшает время выполнения отдельной команды, она повышает пропускную способность процессора, что приводит к более быстрому выполнению программы по сравнению с простой, не конвейерной схемой [1].

1.2 Разреженный строчный формат матрицы

Разреженный строчный формат (сокр. РСФ) — это одна из наиболее широко используемых схем хранения разреженных матриц [2].

Значения ненулевых элементов матрицы и соответствующие столбцовые индексы хранятся в этой схеме по строкам в двух массивах; назовем их соответственно AN и JA. Используется также массив указателей (скажем, IA, еще обозначающийся как NR), отмечающих позиции массивов AN и JA, с которых начинается описание очередной строки. Дополнительная компонента в IA содержит указатель первой свободной позиции в JA и AN.

Рассмотрим матрицу A:

$$A = \begin{pmatrix} 0 & 0 & 0 & 1. & 3. & 0 & 0 & 5. & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7. & 0 & 1. & 0 & 0 \end{pmatrix} \tag{1.1}$$

Разреженный строчный формат матрицы 1.1) представлен ниже:

позиция	=	1	2	3	4	5	6
AN	=	1.	3.	5.	7.	1.	
JA	=	3	4	8	6	8	
NR	=	1	4	4	6		

В общем случае описание i-й строки хранится в позициях с $IA(i)$ до $IA(i+1)-1$ массивов JA и AN, за исключением равенства $IA(i+1)=IA(i)$, означающего, что r-я строка пуста. Если матрица имеет n строк, то IA содержит $n+1$ позиций.

1.3 Сложение РСФ матриц

Для сложения матриц в РСФ необходимо пройти через матрицу указателей IA поэлементно через две матрицы, выделяя интервалы между соседними элементами IA по AN и JA, то есть $IA(i)$ и $IA(i+1)$.

Если встретится элементы с одинаковыми строками и столбцами, то производится сложение. Затем происходит повторные проходы по двум матрицам по отдельности для того чтобы добавить в результирующую матрицу остаточные значение [2].

1.4 Описание алгоритма

В качестве операций, выполняющихся на конвейере, взяты следующие:

1) создание двух матриц в РСФ;

2) сложение двух созданных ранее матриц в РСФ;

3) распаковка результата РСФ в классическое матричное представление.

Ленты конвейера (обработчики) будут передавать друг другу заявки. Первый этап, или обработчик, будет формировать заявку, которая будет передаваться от этапа к этапу

Заявка будет содержать:

— две матрицы в РСФ;

— результат сложения двух матриц в РСФ;

— матрица в классическом представлении для распаковки;

— временные отметки начала и конца выполнения стадии обработки заявки.

Вывод

В данном разделе было рассмотрено понятие конвейерной обработки, а также выбраны этапы для обработки матрицы, которые будут обрабатывать ленты конвейера. Также рассмотрена разреженная матрицы и операция сложения таких матриц.

Программа будет получать на вход количество задач, размеры матрицы и количество ненулевых элементов, а также выбор алгоритма — линейный или конвейерный.

2 Конструкторская часть

В данном разделе будут представлены схемы последовательной и параллельной работы стадий конвейера.

2.1 Требования к программному обеспечению

К программе предъявлен ряд требований:

— наличие меню для выбора запускаемого режима работы конвейера — последовательного и параллельного;

— предоставление интерфейса для ввода линейного размера обрабатываемых матриц и числа заявок;

— формирование файла с логом работы конвейера, логирование событий обработки должно происходить после окончания работы, собственно, конвейера.

2.2 Разработка алгоритмов

На рисунке 2.1 представлен алгоритм генерации РСФ-матрицы.

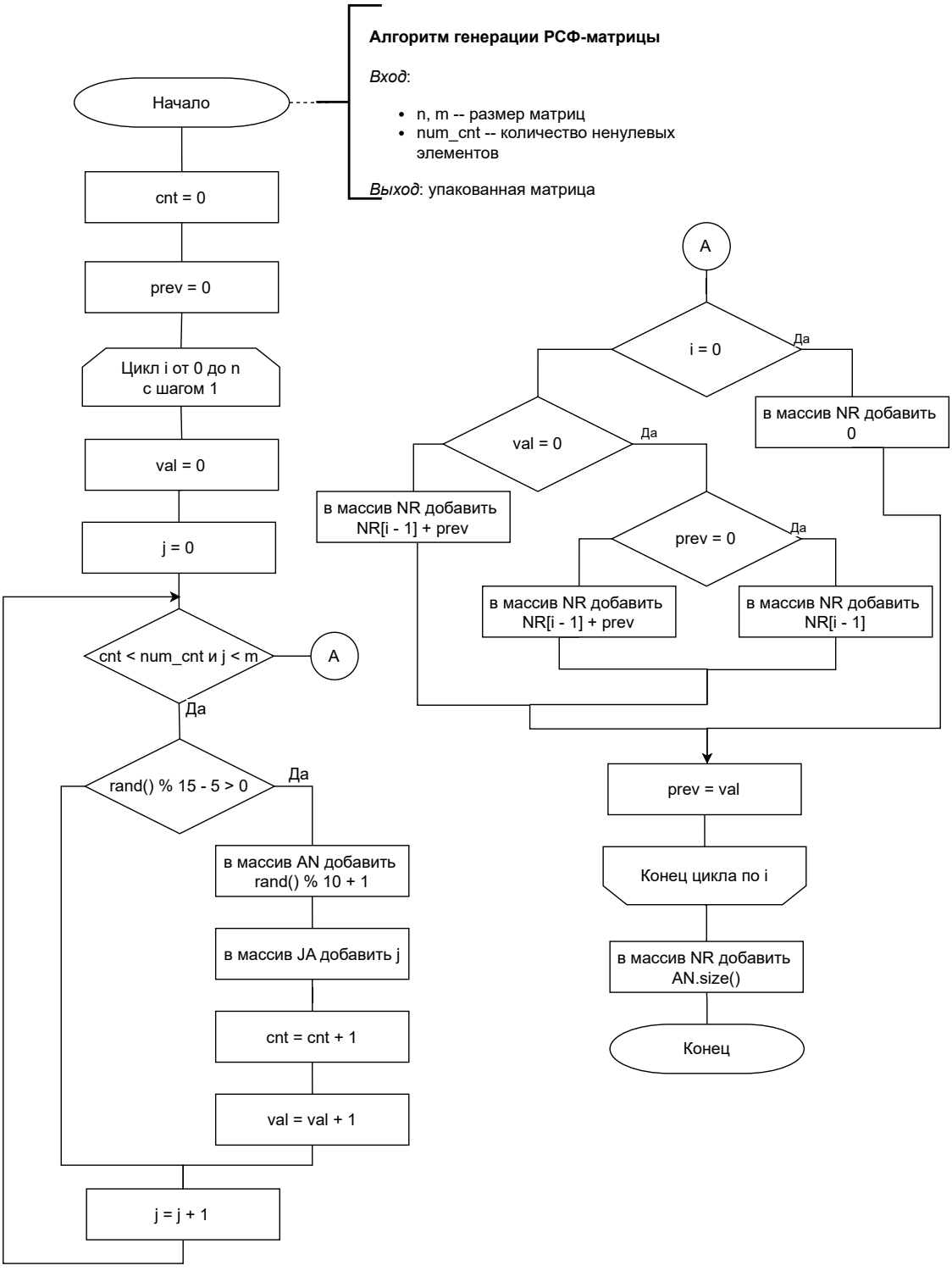

Рисунок 2.1 – Схема алгоритма генерации РСФ-матрицы

На рисунках 2.2 – 2.4 представлен алгоритм сложения РСФ-матриц.

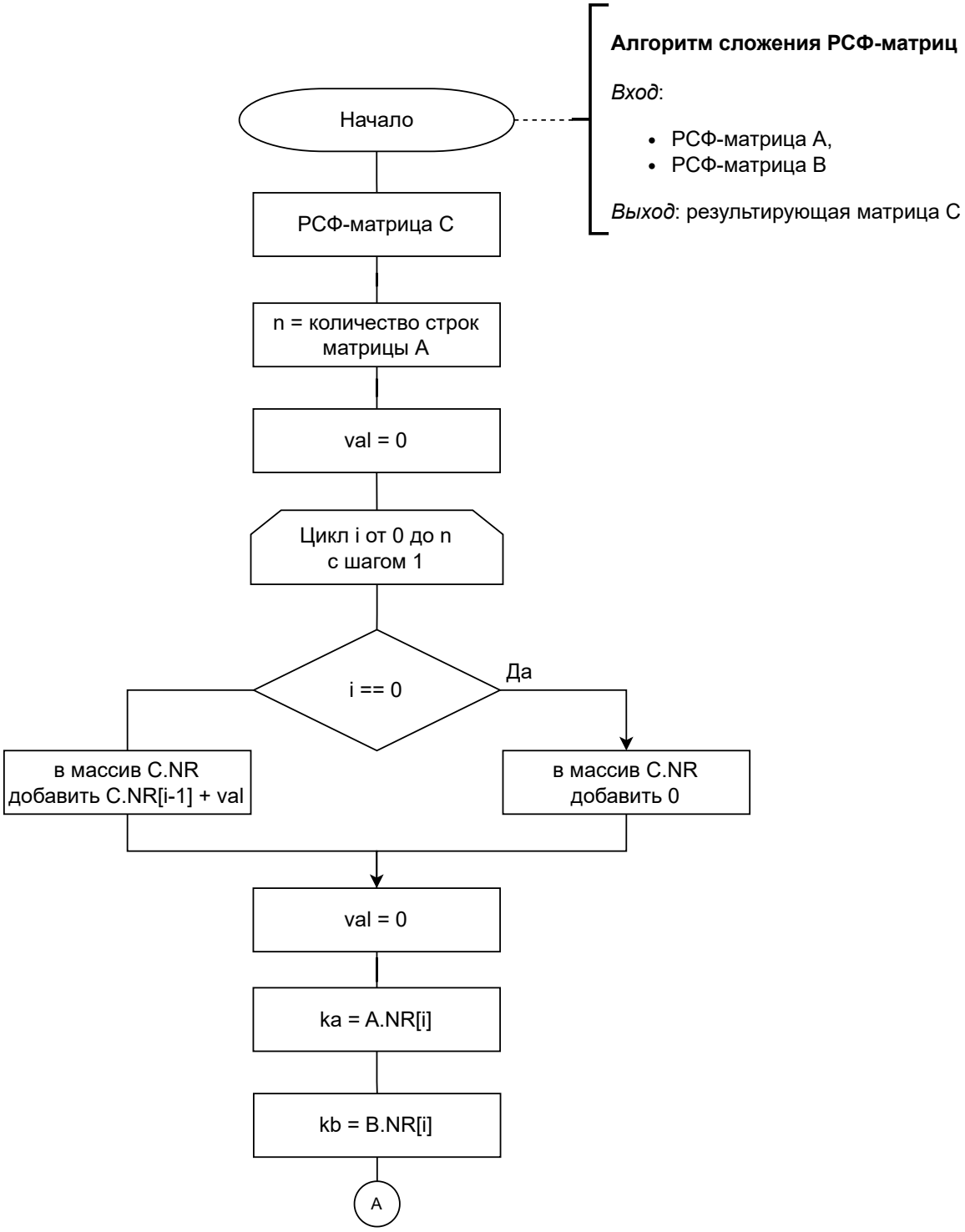

Рисунок 2.2 – Схема алгоритма сложения РСФ-матриц(часть 1)

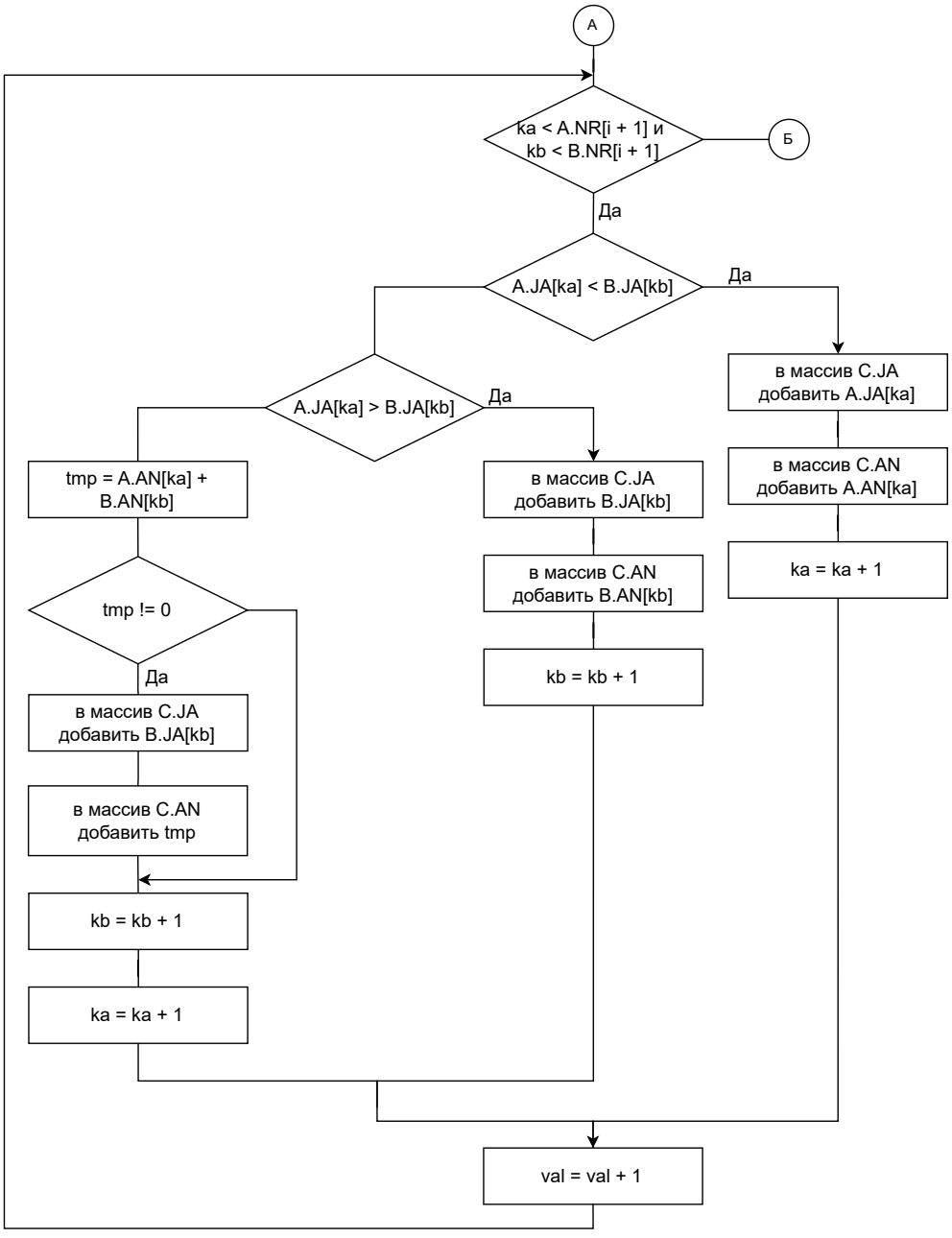

Рисунок 2.3 – Схема алгоритма сложения РСФ-матриц(часть 2)

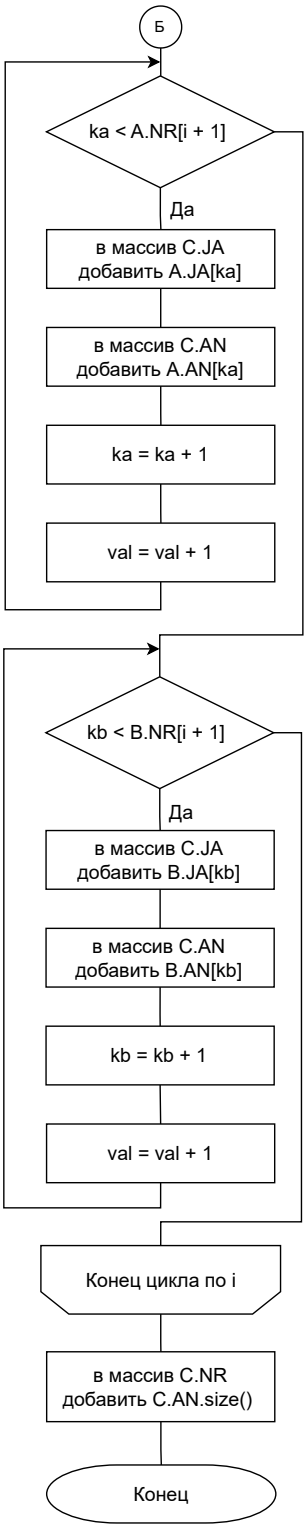

Рисунок 2.4 – Схема алгоритма сложения РСФ-матриц(часть 3)

На рисунке 2.5 представлен алгоритм распаковки РСФ-матрицы.

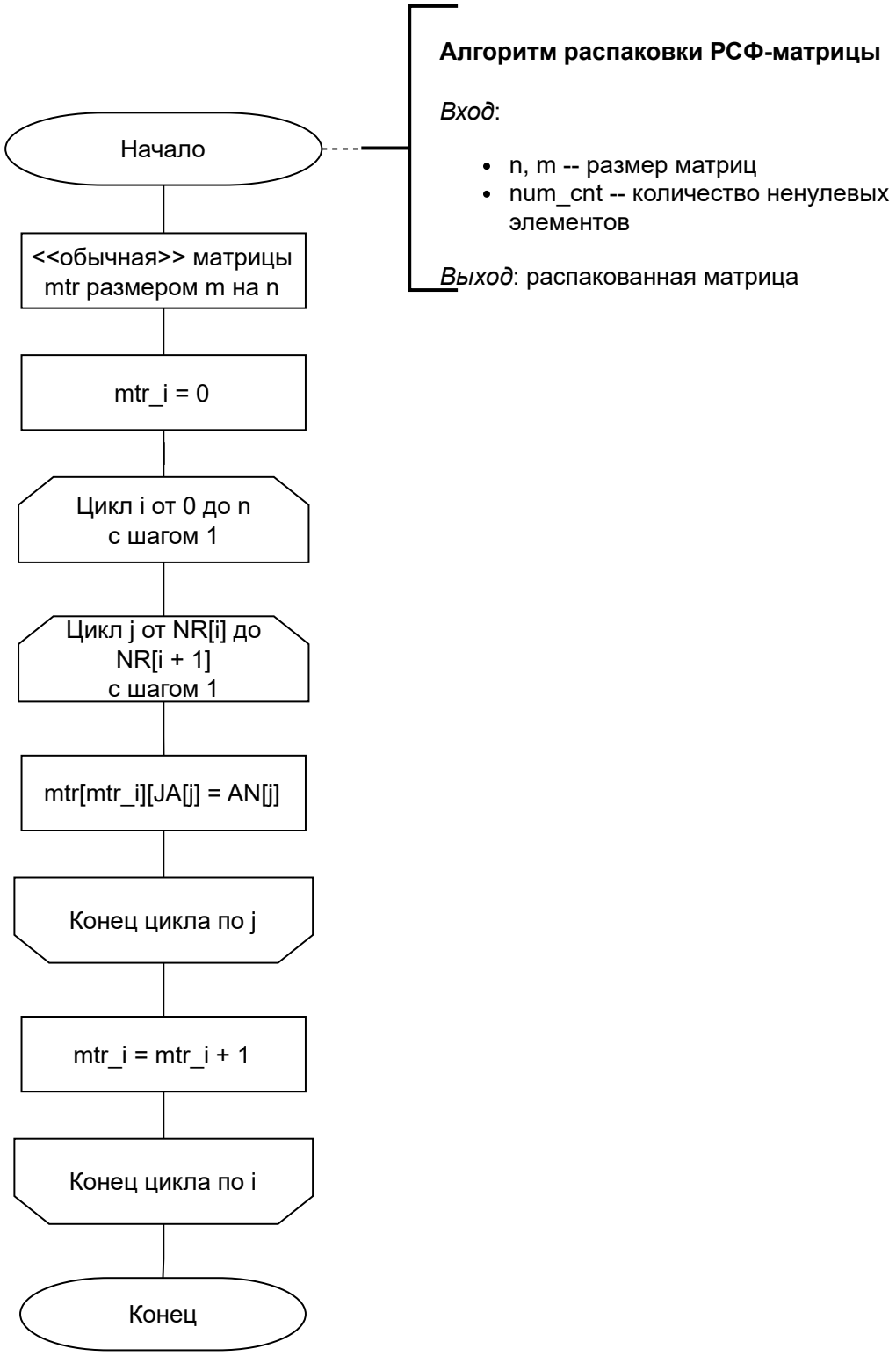

Рисунок 2.5 – Схема алгоритма распаковки РСФ-матрицы

На рисунке 2.6 представлен последовательный алгоритм работы стадий конвейера.

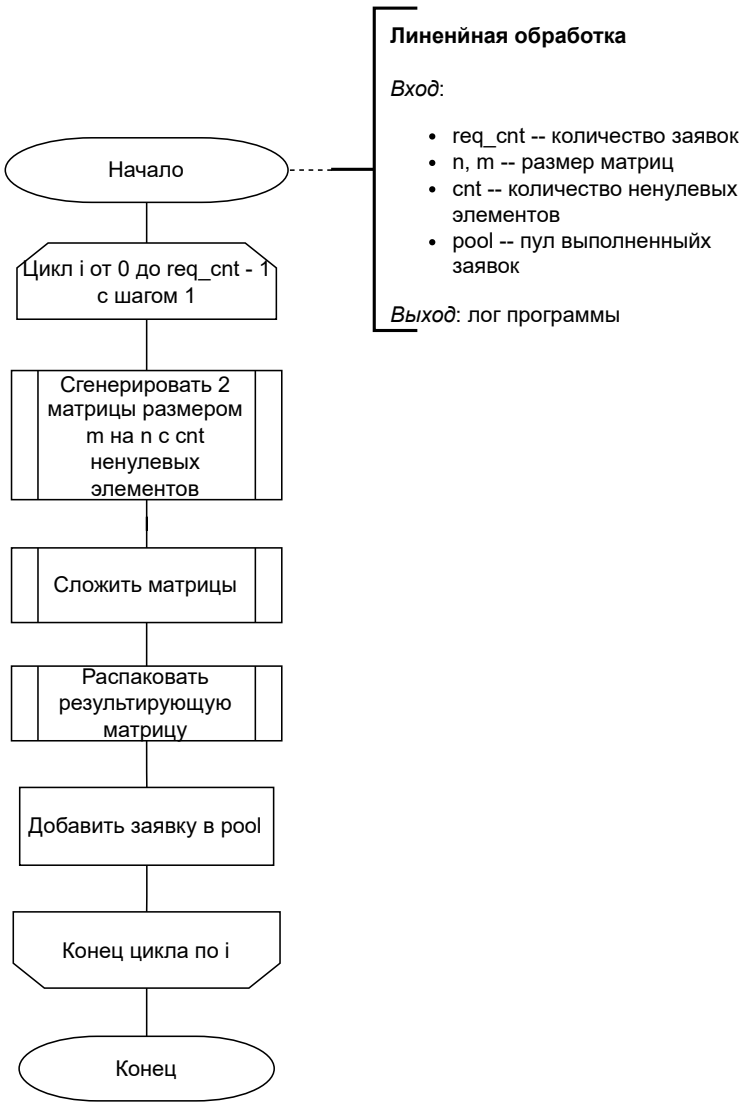

Рисунок 2.6 – Схема последовательной обработки

Параллельная работа будет реализована с помощью добавления 3-х вспомогательных потоков, где каждый поток отвечает за свою стадию обработки. Вспомогательному потоку в числе аргументов в качестве структуры будут переданы:

— две матрицы в РСФ;

— результирующая матрица сложения двух матриц в РСФ;

— временные отметки начала и конца выполнения стадии обработки заявки.

На рисунке 2.7 представлена схема главного потока при параллельной работе стадий конвейера.

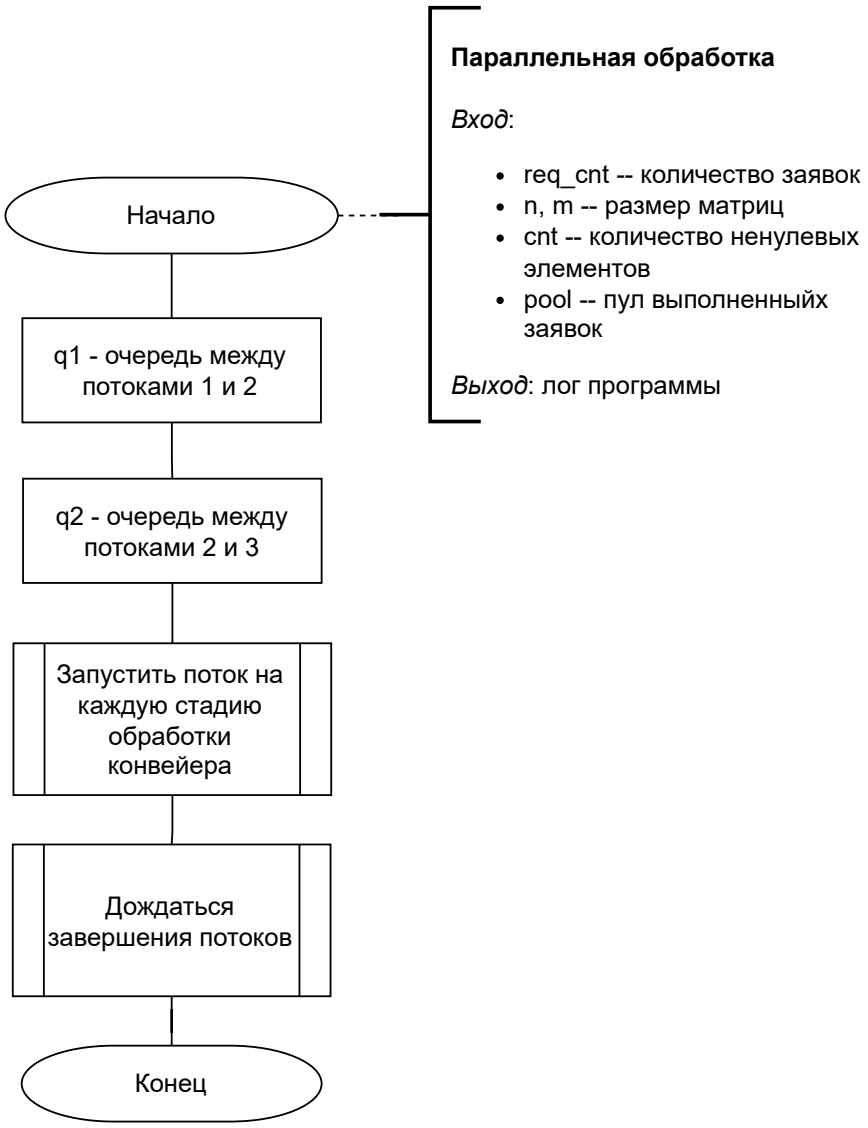

Рисунок 2.7 – Схема параллельной конвейерной обработки

На рисунках 2.8 – 2.10 представлены схемы алгоритмов каждого из обработчиков (потоков) при параллельной работе.

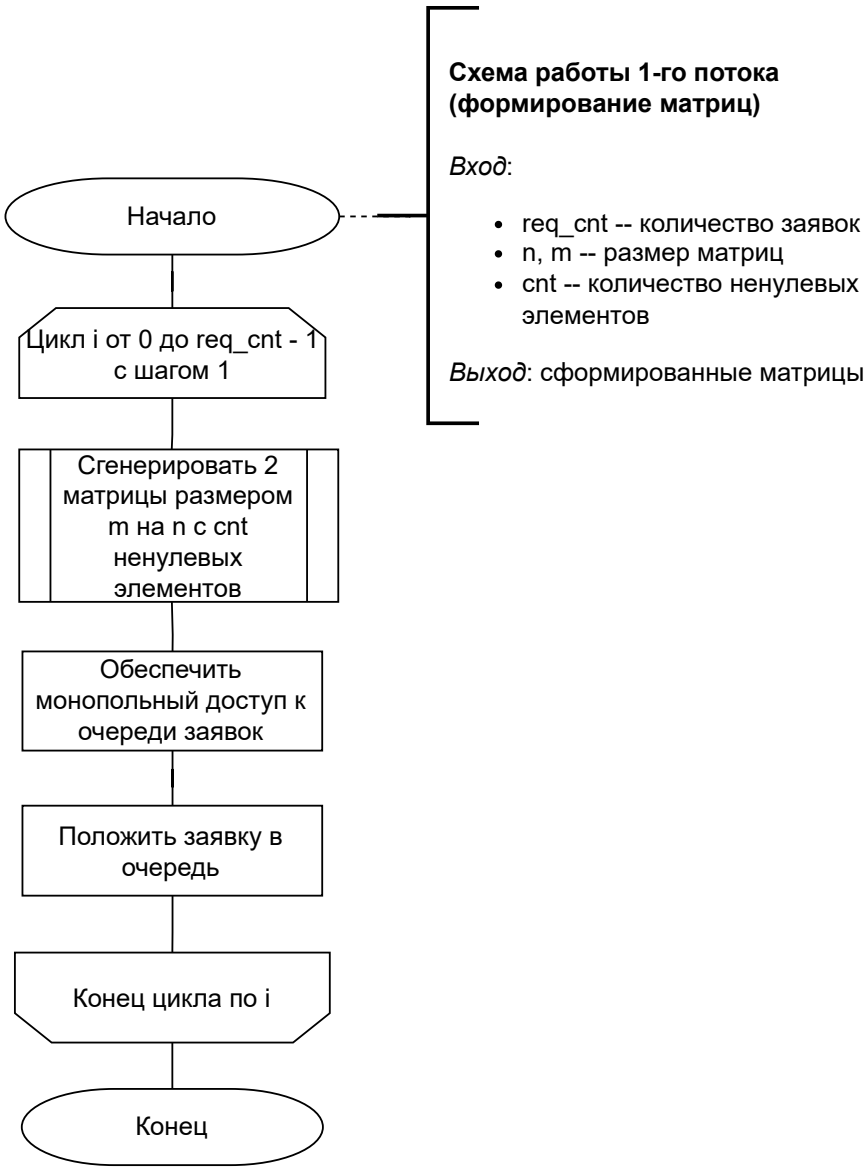

Рисунок 2.8 – Схема алгоритма потока, выполняющего создание и упаковку матриц

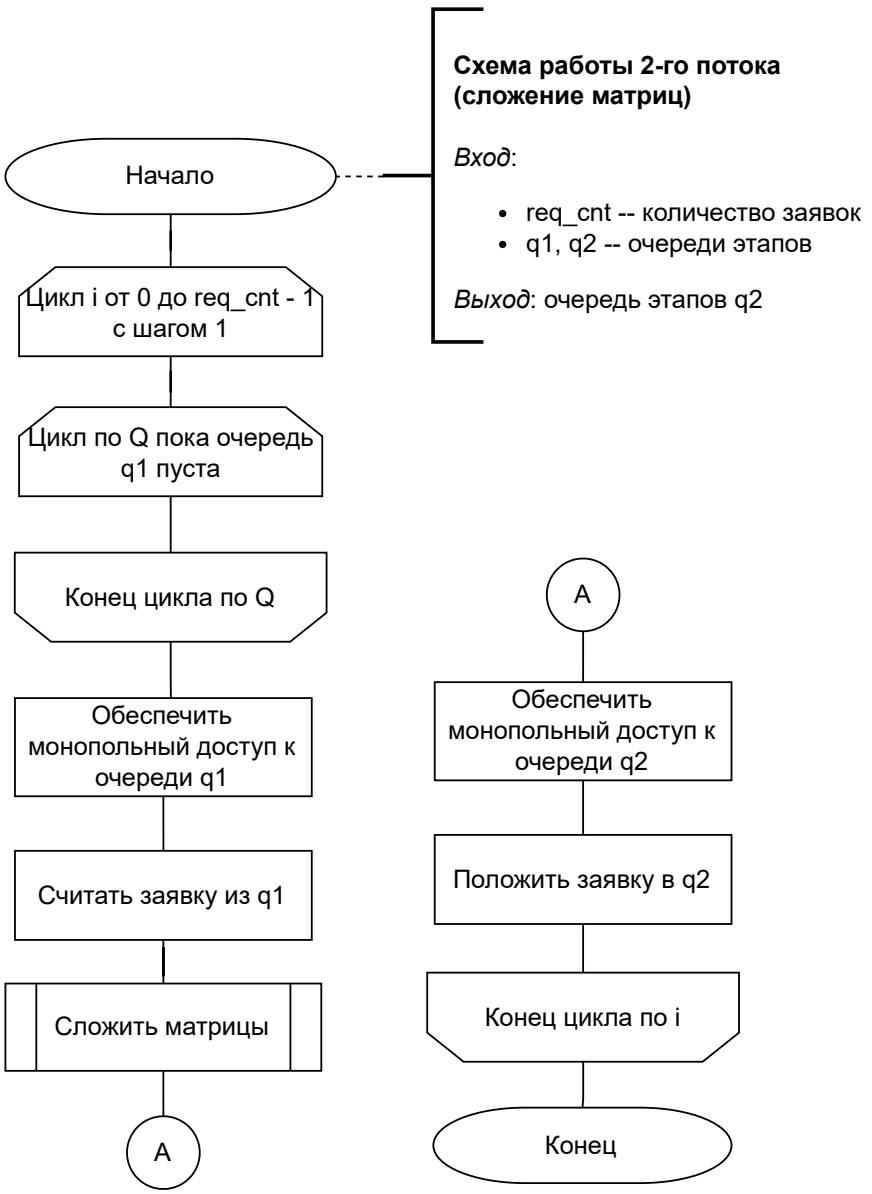

Рисунок 2.9 – Схема алгоритма потока, выполняющего сложение упакованных матриц

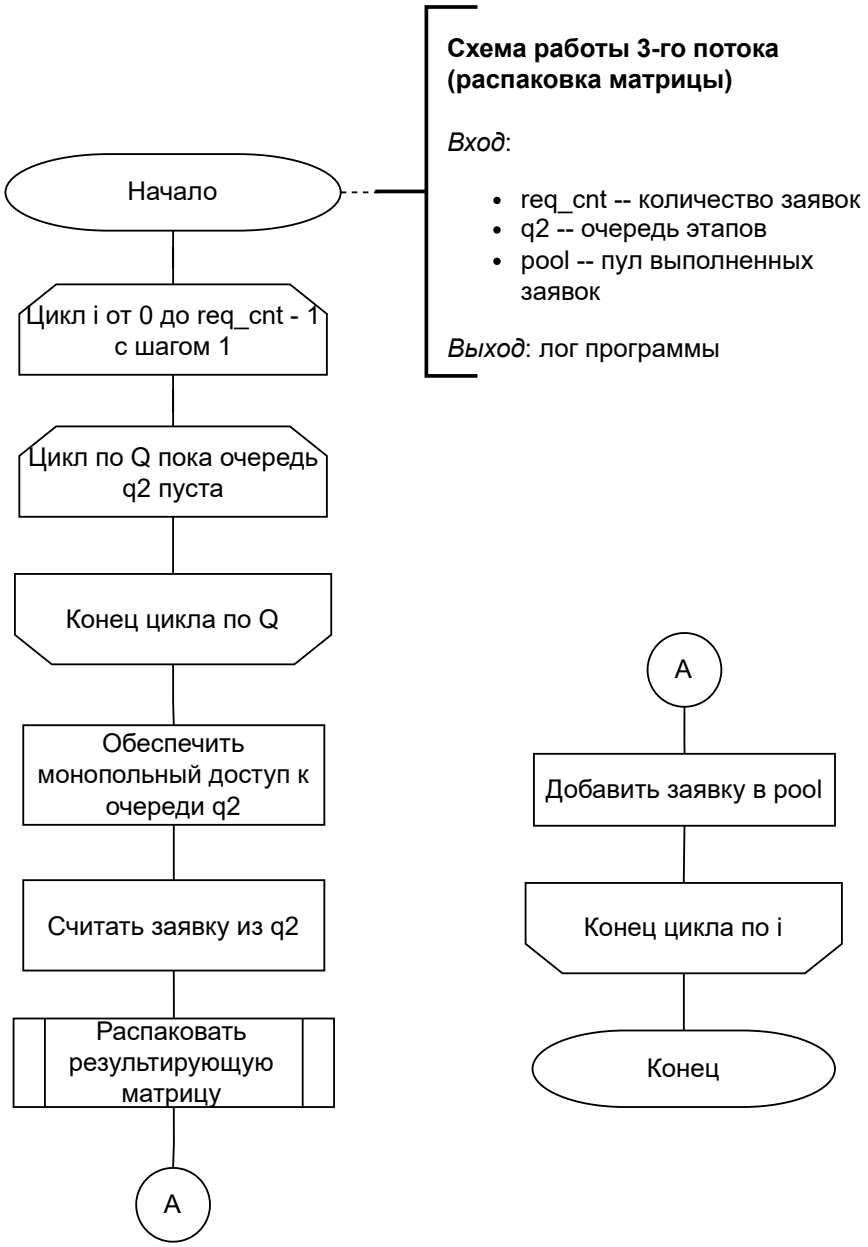

Рисунок 2.10 – Схема алгоритма потока, выполняющего распаковку результирующей матрицы

Вывод

В данном разделе были представлены схемы последовательной и параллельной работы стадий конвейера.

3 Технологический раздел

В данном разделе будут описаны средства реализации программного обеспечения, а также представлены листинги и функциональные тесты.

3.1 Средства реализации

В качестве языка программирования, используемого при написании данной лабораторной работы, был выбран C++ [3], так как в нем имеется контейнер `std::vector`, представляющий собой динамический массив данных произвольного типа, и библиотека `<ctime>` [4], позволяющая производить замеры процессорного времени. Также выбранный язяк программирования предоставляет возможность работы с потоками (класс `thread` [5]) и мьютексами (класс `mutex` [6]).

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

— `main.cpp` — файл, содержащий точку входа в программу;

— `matrix.cpp` — файл содержит функции операций над матрицей и матрицей в РСФ;

— `read.cpp` — файл содержит функции чтения данных;

— `conveyor.cpp` — файл содержит функции конвейерной обработки;

— `meassure.cpp` — файл содержит функции, замеряющее процессорное время работы реализаций алгоритмов.

3.3 Реализация алгоритмов

На листингах 3.1 – 3.8 представлены реализации разрабатываемых алгоритмов.

Листинг 3.1 – Реализация алгоритма генерации данных для матрицы РСФ

```
MatrixCSR::MatrixCSR(size_t _n, size_t _m, size_t num_cnt) {

    int cnt = 0, prev = 0;
    n = _n;
    m = _m;

    for (size_t i = 0; i < n; i++) {

        srand(clock() % 1000000);
        int val = 0;

        for (size_t j = 0; cnt < num_cnt && j < m; j++) {

            if ((rand() % 15) - 5 > 0) {

                AN.push_back(rand() % 10 + 1);
                JA.push_back(j);

                cnt++;
                val++;
            }
        }

        if (i == 0)
            NR.push_back(0);
        else if (val == 0) {
            if (prev == 0)
                NR.push_back(NR[i - 1]);
            else
                NR.push_back(NR[i - 1] + prev);
        }
        else
            NR.push_back(NR[i - 1] + prev);
        prev = val;
    }

    NR.push_back(NR[n - 1] + (AN.size() - NR[n - 1]));
}
```

Листинг 3.2 – Реализация алгоритма сложения РСФ-матриц

```
MatrixCSR MatrixCSR::operator+(const MatrixCSR &mtr) {

    MatrixCSR c;

    c.n = n;
    c.m = n;

    int val = 0;

    for (int i = 0; i < NR.size() - 1; i++) {
        if (i == 0)
            c.NR.push_back(0);
        else
            c.NR.push_back(c.NR[i - 1] + val);
        val = 0;

        int ka = NR[i];
        int kb = mtr.NR[i];

        for (; ka < NR[i + 1] && kb < mtr.NR[i + 1];) {

            if (JA[ka] < mtr.JA[kb]) {
                c.JA.push_back(JA[ka]);
                c.AN.push_back(AN[ka++]);
            } else if (JA[ka] > JA[kb]) {
                c.JA.push_back(mtr.JA[kb]);
                c.AN.push_back(mtr.AN[kb++]);
            } else {
                auto tmp = AN[ka] + mtr.AN[kb];

                if (tmp) {
                    c.JA.push_back(JA[ka]);
                    c.AN.push_back(tmp);
                }
                ka++;
                kb++;
            }
            }
            val++;
        }
```

```
41
42          for (; ka < NR[i + 1]; ka++) {
43              c.JA.push_back(JA[ka]);
44              c.AN.push_back(AN[ka]);
45              val++;
46          }
47
48          for (; kb < mtr.NR[i + 1]; kb++) {
49              c.JA.push_back(mtr.JA[kb]);
50              c.AN.push_back(mtr.AN[kb]);
51              val++;
52          }
53      }
54
55      c.NR.push_back(c.NR[c.NR.size() - 1] + (c.AN.size() -
            c.NR[c.NR.size() - 1]));
56      return c;
57  }
```

Листинг 3.3 – Реализация алгоритма распаковки матрицы

```
MatrixT MatrixCSR::decomprass() {

    MatrixT mtr;

    mtr.m = m;
    mtr.n = n;

    for (int i = 0; i < n; i++ )
    {
        mtr.arr.emplace_back();
        for (int j = 0; j < m; j++ )
            mtr.arr.back().push_back(0);
    }

    int mtr_i = 0;
    for (int i = 0; i < n; i++) {

        for (int j = NR[i]; j < NR[i + 1]; j++)
            mtr.arr[mtr_i][JA[j]] = AN[j];
        mtr_i++;
    }

    return mtr;
}
```

Листинг 3.4 – Реализация последовательной конвейерной обработки

```
void linear() {

    int req_cnt = getRequestNumber();
    int n = getMatrixN();
    int m = getMatrixM();
    int cnt = getMatrixNum();

    vector<requestT *> pool(req_cnt);

    for (int i = 0; i < req_cnt; i++) {

        requestT *r = new requestT();

        clock_gettime(CLOCK_REALTIME, &r->p1_start);
        packData(n, m, cnt, r);
        clock_gettime(CLOCK_REALTIME, &r->p1_end);

        clock_gettime(CLOCK_REALTIME, &r->p2_start);
        r->mtr_c = r->mtr_a + r->mtr_b;
        clock_gettime(CLOCK_REALTIME, &r->p2_end);

        clock_gettime(CLOCK_REALTIME, &r->p3_start);
        r->result = r->mtr_c.decomprass();
        clock_gettime(CLOCK_REALTIME, &r->p3_end);

        pool[i] = r;
    }

    printPool(pool, "linear.txt");

    for (size_t i = 0; i < pool.size(); ++i)
        delete pool[i];
}
```

Листинг 3.5 – Реализация основного потока для конвейерной обработки, создающий вспомогательные потоки

```
void parallel() {

    int req_cnt = getRequestNumber();
    int n = getMatrixN();
    int m = getMatrixM();
    int cnt = getMatrixNum();

    vector<requestT *> pool(req_cnt);
    queue<requestT *> q1;
    queue<requestT *> q2;

    thread t_1(thread_1, req_cnt, n, m, cnt, ref(q1));
    thread t_2(thread_2, req_cnt, ref(q1), ref(q2));
    thread t_3(thread_3, req_cnt, ref(q2), ref(pool));

    t_1.join();
    t_2.join();
    t_3.join();

    printPool(pool, "parallel.txt");
    for (size_t i = 0; i < pool.size(); ++i)
        delete pool[i];
}
```

Листинг 3.6 – Реализация вспомогательного потока, отвечающий за создание матриц РСФ

```
void thread_1(size_t req_cnt, size_t n, size_t m, size_t cnt,
   queue<requestT *> &q1) {

    for (int i = 0; i < req_cnt; i++) {

        requestT *r = new requestT();

        clock_gettime(CLOCK_REALTIME, &r->p1_start);
        packData(n, m, cnt, r);

        mutex_q1.lock();
        clock_gettime(CLOCK_REALTIME, &r->p1_end);
        q1.push(r);
        mutex_q1.unlock();
    }
}
```

Листинг 3.7 – Реализация вспомогательного потока, отвечающий за сложение матриц РСФ

```
1  void thread_2(int req_cnt, queue<requestT *> &q1, queue<requestT
       *> &q2) {
2
3      for (int i = 0; i < req_cnt; i++) {
4
5          while (q1.empty());
6
7          mutex_q1.lock();
8          requestT *r = q1.front();
9          q1.pop();
10         mutex_q1.unlock();
11
12         clock_gettime(CLOCK_REALTIME, &r->p2_start);
13         r->mtr_c = r->mtr_a + r->mtr_b;
14
15         mutex_q2.lock();
16         clock_gettime(CLOCK_REALTIME, &r->p2_end);
17         q2.push(r);
18         mutex_q2.unlock();
19     }
20 }
```

Листинг 3.8 – Реализация вспомогательного потока, отвечающий за распаковку матрицы

```
void thread_3(int req_cnt, queue<requestT *> &q2,
   vector<requestT *> &pool) {

    for (int i = 0; i < req_cnt; i++) {

        while (q2.empty());

        mutex_q2.lock();
        requestT *r = q2.front();
        q2.pop();
        mutex_q2.unlock();

        clock_gettime(CLOCK_REALTIME, &r->p3_start);
        r->result = r->mtr_c.decomprass();
        clock_gettime(CLOCK_REALTIME, &r->p3_end);
        pool[i] = r;
    }
}
```

Вывод

В данном разделе была приведена информация о выбранных средствах для разработки алгоритмов. Были представлены листинги для каждой из реализаций работы конвейера и его трех стадий, а именно генерации данных для двух матриц РСФ, сложение двух матриц РСФ и распаковка матрицы РСФ в классическое матричное представление.

4 Исследовательский раздел

В данном разделе будут приведены примеры работы программ, постановка исследования и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени, представлены далее.

— процессор: AMD Ryzen 5 5500U – 2.10 ГГц;

— Оперативная память: 16 ГБайт;

— Операционная система: Windows 10 Pro 64-разрядная система версии 22H2.

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен пример результата работы программы. Пользователь, указывая соответствующие пункты меню, запускает последовательную обработку заявок, затем параллельное исполнение конвейера, затем выходит из программы.

Лог программы при этом записывается в файл. На рисунке 4.2 представлен пример лог–файла.

```
Выберите необходимую задачу:
1 - запустить последовательную обработку матриц
2 - запустить конвейерную обработку матриц
3 - замеры времени реализаций
0 - выход

Ваш выбор: 1
Введите количество заявок (больше 0): 10
Введите количество строк n (больше 0): 10
Введите количество столбцов m (больше 0): 10
Введите количество элементов (больше 0): 10
```

Рисунок 4.1 – Демонстрация работы программы

```
Request 0 start creating: 0 ns
Request 0 end creating: 34200 ns
Request 0 start sum: 34200 ns
Request 0 end sum: 49400 ns
Request 0 start unpack: 49500 ns
Request 0 end unpack: 80100 ns
Request 1 start creating: 80500 ns
Request 1 end creating: 94400 ns
Request 1 start sum: 94500 ns
Request 1 end sum: 101200 ns
Request 1 start unpack: 101200 ns
Request 1 end unpack: 127000 ns
Request 2 start creating: 127400 ns
Request 2 end creating: 148900 ns
Request 2 start sum: 148900 ns
Request 2 end sum: 155500 ns
Request 2 start unpack: 155500 ns
Request 2 end unpack: 181000 ns
Request 3 start creating: 181500 ns
Request 3 end creating: 197100 ns
Request 3 start sum: 197100 ns
Request 3 end sum: 203700 ns
Request 3 start unpack: 203800 ns
```

Рисунок 4.2 – Пример файла с логом работы конвейера

4.3 Временные характеристики

Для замеров времени использовалась функция получения значения системных часов `clock_gettime()` [4]. Функция применялась два раза — в начале и в конце измерения времени, значения полученных временных меток вычитались друг из друга для получения времени выполнения программы.

Исследование временных характеристик реализаций алгоритмов производилось при изменении числа заявок от 10 до 100 с шагом 10 для матриц размером 10.

В таблице 4.1 представлены замеры времени выполнения двух реализаций конвейерной обработки в зависимости от количества заявок:

Таблица 4.1 – Результаты нагрузочного тестирования (в мкс)

Кол-во заявок	**Время, мкс**	
	Последовательный	**Параллельный**
10	340.06	476.88
20	688.36	749.03
30	1002.81	965.46
40	1251.81	1272.91
50	1603.30	1658.57
60	2055.12	2152.98
70	2402.26	2431.97
80	2754.30	2729.88
90	3057.35	2843.42
100	3449.13	3076.39

На рисунке 4.3 приведен график результатов замеров для различных значений количества заявок.

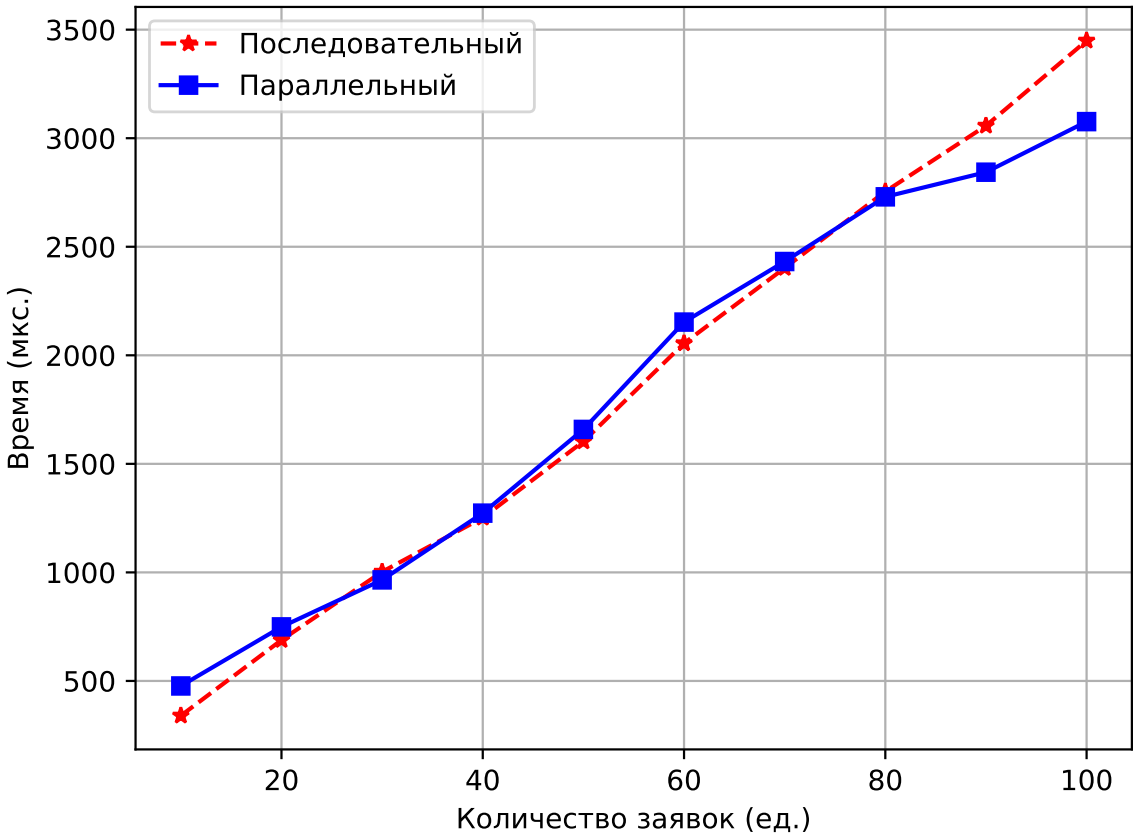

Рисунок 4.3 – Результаты замеров времени работы реализации конвейерной обработки при варьировании числа заявок

4.4 Вывод

В результате эксперимента было получено, что использование конвейерной обработки лучше по времени линейной реализации на 100 заявках примерно в 1.2 раза.

В силу линейности графиков на рисунке 4.3 можно сказать, что на достаточно большом количестве заявок выигрыш параллельной обработки над последовательной во времени в абсолютных единицах будет увеличиваться.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы было выявлено, что в результате использования параллельной реализации конвейерной обработки примерно в 1.2 раза лучше работает, чем последовательная конвейерная обработка.

Цель, поставленная в начале работы, была достигнута. Кроме того были достигнуты все поставленные задачи:

1) описана организация конвейерной обработки данных;

2) описаны алгоритмы обработки данных, которые будут использоваться в текущей лабораторной работе;

3) реализована программа, выполняющую конвейерную обработку с количеством лент не менее трех в однопоточной и многопоточной реализаций;

4) проведены сравнение и анализ реализации алгоритмов по затраченному времени.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Конвейерная обработка данных. — [Электронный ресурс]. — Режим доступа: https://studfile.net/preview/1083252/page:25/ (дата обращения: 21.12.2023).
2. *С. П.* Технология разреженных матриц //. — — М.: Издательство «МИР», 1998. — С. 410.
3. C++ language. — [Электронный ресурс]. — Режим доступа: https://en.cppreference.com/w/cpp/language (дата обращения: 21.12.2023).
4. Standard library header <ctime>. — [Электронный ресурс]. — Режим доступа: https://en.cppreference.com/w/cpp/thread/thread (дата обращения: 21.12.2023).
5. std::thread. — [Электронный ресурс]. — Режим доступа: https://en.cppreference.com/w/cpp/thread/thread (дата обращения: 22.12.2023).
6. std::mutex. — [Электронный ресурс]. — Режим доступа: https://en.cppreference.com/w/cpp/thread/mutex (дата обращения: 22.12.2023).