Анализ Алгоритмов

Лабораторная работа №4

«Параллельное умножение матриц»

Юмаев Артур Русланович

ИУ7-55

Преподаватель: Волкова Л.Л.

Оглавление

[Введение 3](#_Toc24909754)

[Постановка задачи 4](#_Toc24909755)

[1. Аналитическая часть 5](#_Toc24909756)

[Классический подход 5](#_Toc24909757)

[Алгоритм Винограда 6](#_Toc24909758)

[Оптимизированный алгоритм Винограда 6](#_Toc24909759)

[Распараллеливание задачи на CPU 7](#_Toc24909760)

[2. Конструкторская часть 8](#_Toc24909761)

[Оценка трудоемкости алгоритмов 8](#_Toc24909762)

[Классический алгоритм умножения матриц 8](#_Toc24909763)

[Алгоритм Винограда 8](#_Toc24909764)

[Оптимизированный алгоритм Винограда 9](#_Toc24909765)

[Схемы алгоритмов 10](#_Toc24909766)

[3. Технологическая часть 13](#_Toc24909767)

[4. Исследовательская часть 15](#_Toc24909768)

[Заключение 16](#_Toc24909769)

# Введение

Алгоритмы перемножение матриц имеют довольно обширное применение. В данное время они используются в области искусственного интеллекта. Перемножение матриц – одна из основных операций, на которых строятся алгоритмы искусственных нейронных сетей, которые могут анализировать изображения, находить скрытые взаимосвязи и классифицировать их, заранее обучившись на некой тренировочной выборки. Перемножения матриц также используются в прикладной физике, математике, математической статистике и многих других прикладных науках.

# Постановка задачи

Цель: изучение алгоритмов умножения матриц. В данной работе рассматривается распараллеленная реализация стандартного алгоритма умножения матриц, алгоритма Винограда.

Задачи:

* Рассматреть задачу умножения матриц;
* Произвести анализ сложности алгоритмов умножения матриц;
* Запрограммировать алгоритмы и сделать их работу параллельной;
* Сделать замеры времени для алгоритмов;
* Результаты экспериментов сравнить с теоретическими оценками трудоёмкости;
* Сделать выводы.

# Аналитическая часть

В данный момент существуют несколько алгоритмов перемножения матриц. Ниже в таблице 1 приведен их список с коэффициентом ω, который показывает сложность алгоритмов ).

Таблица 1

Сравнение эффективности по времени алгоритмов умножения матриц

|  |  |
| --- | --- |
| Алгоритм | ω |
| Классический (1950) | 3.0 |
| Алгоритм Пана (1978) | 2.78041 |
| Алгоритм Бини (1979) | 2.7799 |
| Алгоритм Шёнхаге (1981) | 2.695 |
| Алгоритм Копперсмита — Винограда (1990) | 2.3727 |

Алгоритм, который будет реализован в данной работе является одним из самых эффективных на данный момент.

В данном разделе будет приведено математическое описание алгоритмов перемножения матриц. Буду рассмотрены 3 подхода: классический, алгоритм Винограда и оптимизированный алгоритм Винограда.

## Классический подход

Предположим, что необходимо получить матрицу Для нахождения значений элементов матрицы используют следующее выражение:

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

Классический алгоритм напрямую реализует эту формулу.

## Алгоритм Винограда

Можно заметить, что элементы из суммы выражения 1 можно переписать как:

т. е. как сумму произведения сумм и двух произведений. Учитывая, что упомянутые два произведения можно рассчитать заранее для обработки двух элементов матрицы теперь нужно не сложение и два умножения, а умножение и два сложения, что проще с точки зрения вычислений. Таким образом, алгоритм Винограда состоит в следующем.

1. Совершить расчет заранее двух произведений для каждого ряда и столбца матрицы-результата (одно произведение считается для ряда, другое для столбца). Для хранения результатов используется промежуточный буфер;
2. По вышеприведённой формуле осуществить расчёт каждого элемента матрицы;
3. В случае, если в произведении b – нечётное число, пройтись во второй раз по матрице, дополняя элементы недостающим элементом (который не был описан вышеописанной суммой).

Можно заметить, что пункт 3 необходимо выполнять только в некоторых случаях, но если это происходит, то получается существенное увеличение времени работы алгоритма.

## Оптимизированный алгоритм Винограда

1. Внутри тройного цикла накапливать результат в буфер, а вне цикла сбрасывать его в ячейку матрицы.
2. Заранее вычислить d = [n / 2], где [x] – целая часть x,  
   где n = b в матрице B(b, c)
3. Заменить MulH[i] = MulH[i] + … на MulH[i] += … (аналогично для MulV),  
   где MulH и MulB – временные массивы для предварительного рассчета сумм произведений

## Распараллеливание задачи на CPU

В рамках данной лабораторной работы производилось распараллеливание задачи по потокам. В CPU для данной цели используются threads.

Central Processing Unit (CPU) – это универсальный процессор, также именуемый роцессором общего назначения. Он оптимизирован для достижения высокой производительности единственного потока команд. Доступ к памяти с данными и инструкциями происходит преимущественно случайным образом.

Для того, чтобы повысить производительность CPU ещё больше, они проектируются специально таким образом, чтобы выполнять как можно больше инструкций параллельно. Например для этого в ядрах процессора используется блок внеочередного выполнения команд.

Но несмотря на это, CPU всё равно не в состоянии осуществить параллельное выполнение большого числа инструкций, так как расходы на распараллеливание инструкций внутри ядра оказываются очень существенными. Именно поэтому процессоры общего назначения имеют не очень большое количество исполнительных блоков.

# 2. Конструкторская часть

## Оценка трудоемкости алгоритмов

Используется C-подобная модель оценки трудоёмкости.

Трудоёмкость операций:

1. Трудоемкость операций: +, −, =, + =, − =, <, > ==, ++ равна 1.
2. Трудоемкость операций: \*, /, % равна 2.
3. Трудоемкость операции доступа к элементу памяти: [...] равна 3.

Трудоемкость смены ячеек памяти местами будем считать 9, так как производится 3 обращения к памяти.

Цикл будет оцениваться о фактически выполненным операциям из перечня выше.

Условный оператор if будет фактически оценен как сумма стоимости операций в условии и трудоемкости различных ветвей (в лучшем случае и в худшем случае). Стоимость условного перехода из уловия в одну из ветвей решения полагается равной 0.

### Классический алгоритм умножения матриц

Итоговая формула:

### Алгоритм Винограда

Цикл №1

Цикл №2

Цикл №3

Условие четности/нечетности

Итоговая формула получается из суммы вышеприведенных формул.

### Оптимизированный алгоритм Винограда

Итоговая формула для лучшего случая:

Цикл №1

Цикл №2

Цикл №3

Условие четности/нечетности

Итоговая формула получается из суммы вышеприведенных формул.

## Схемы алгоритмов

В данном разделе будут приведены схема алгоритма Винограда и схема функция распараллеливания. На рисунке 1 изображена схема функции распараллеливания.

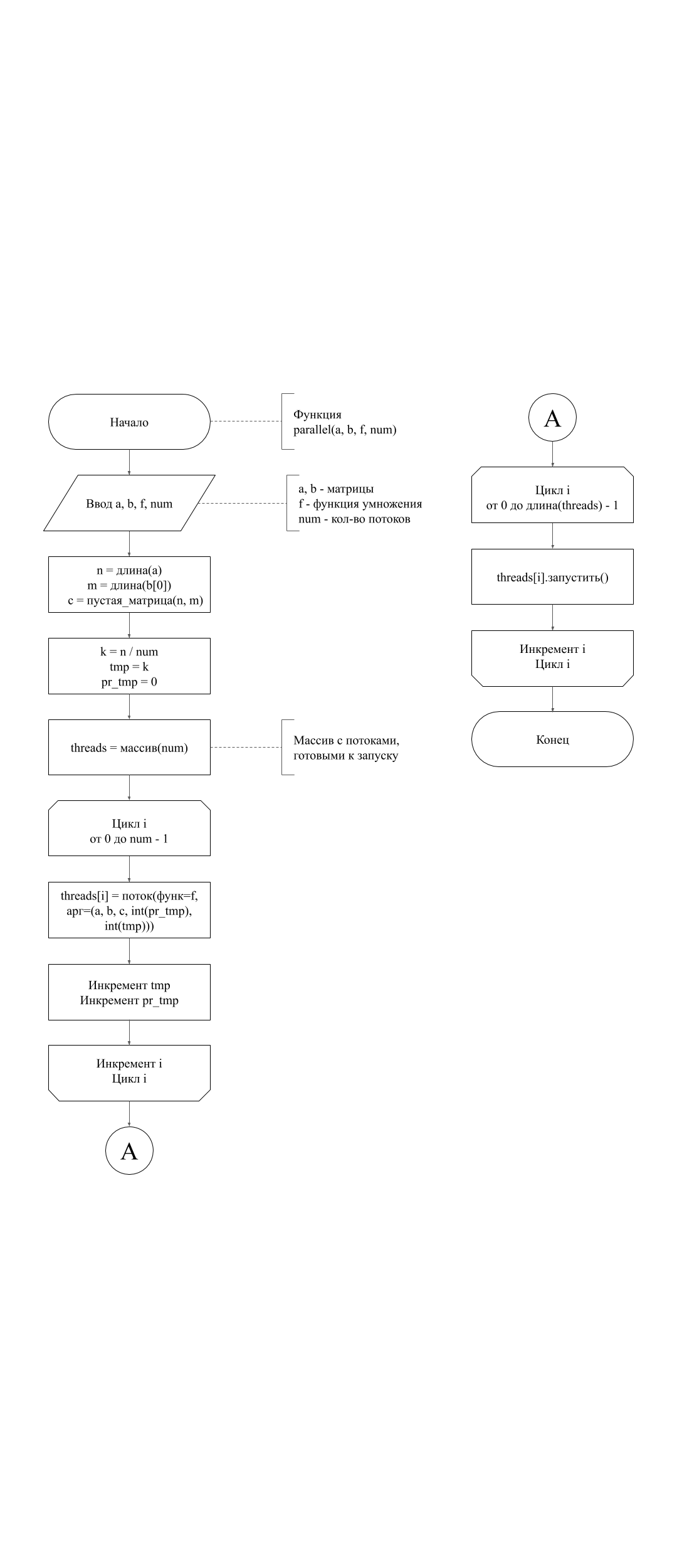


Рисунок 1. Схема фукции распараллеливания. Часть А

На рисунке 2 изображена схема распараллеленного алгоритма Винограда. Часть 1.

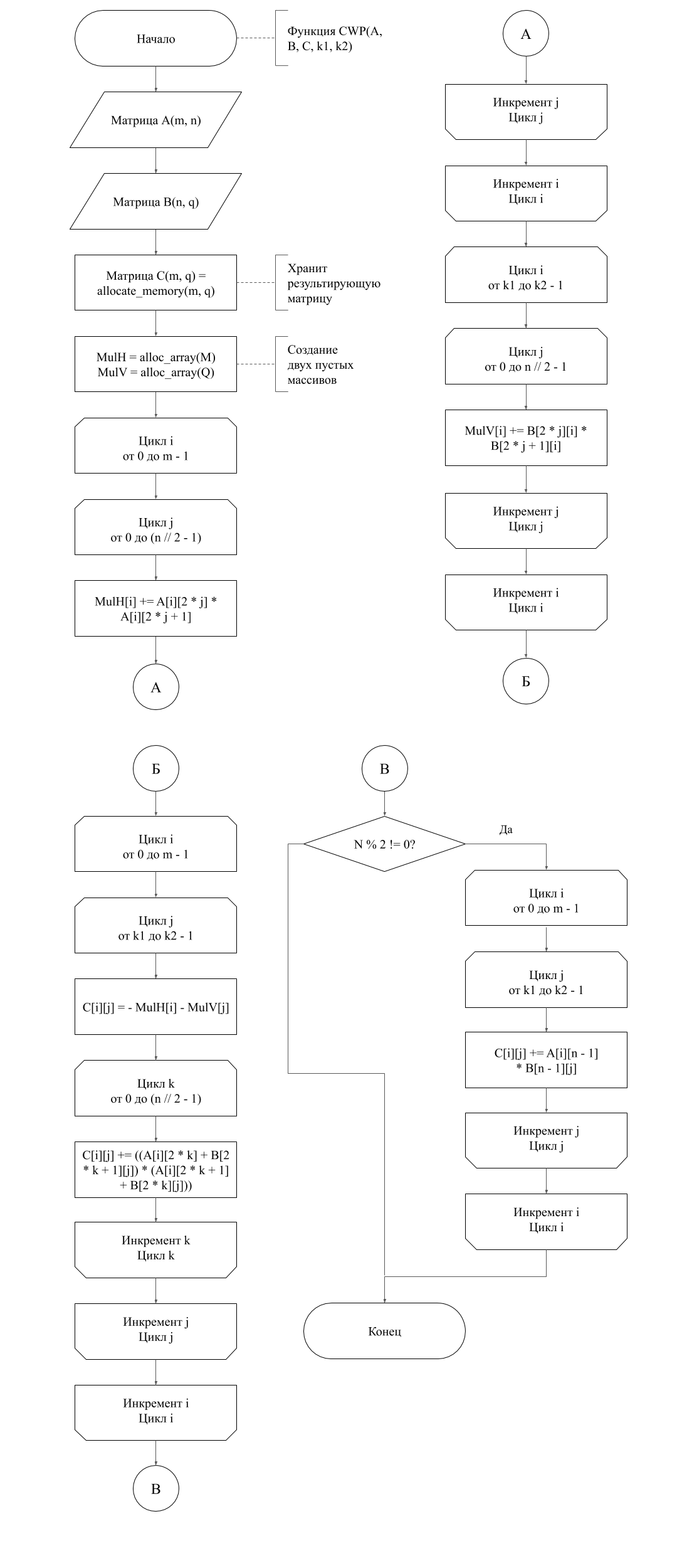


Рисунок 2. Распараллеленный алгоритм Винограда. Часть 1.

На рисунке 3 изображена схема распараллеленного алгоритма Винограда. Часть 2.

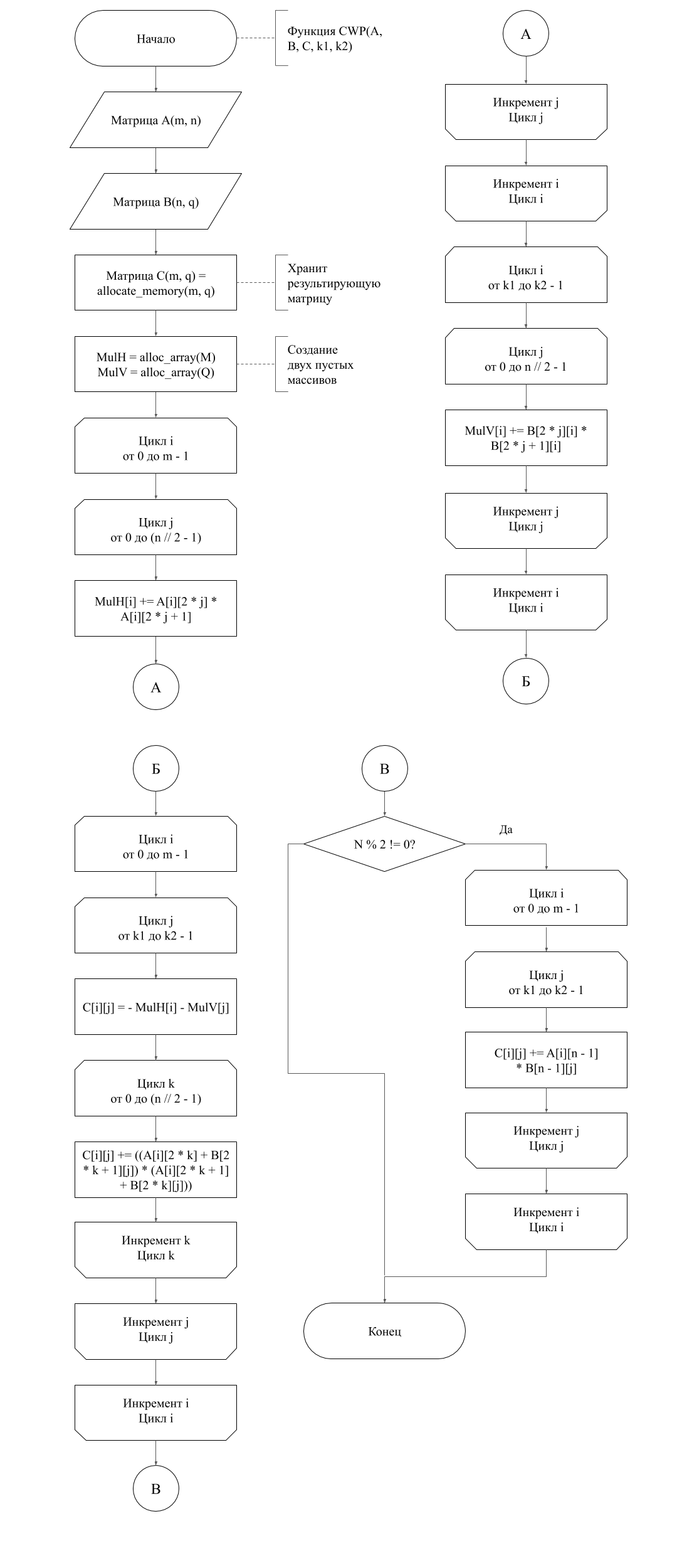


Рисунок 3. Распараллеленный алгоритм Винограда. Часть 2.

# 3. Технологическая часть

В данном разделе будут приведены листинги для каждого из алгоритмов на языке Python (листинги 1 – 3).

Листинг 1. Классический алгоритм умножения матриц

|  |
| --- |
| def classic\_matrix\_mult(A, B):  A, B = np.array(A), np.array(B)    if len(B) != len(A[0]):  print("Error! Different dimension!")  return None  n = len(A)  m = len(A[0])  t = len(B[0])  C = np.zeros((A.shape[0], B.shape[1]))  for i in range(n):  for j in range(m):  for k in range(t):  C[i][k] += A[i][j] \* B[j][k]  return C |

Листинг 2. Алагоритм Винограда с поддержкой параллельности

|  |
| --- |
| def classic\_winograd\_mult(A, B, C, k1, k2):  M = len(A)  N = len(B)  Q = len(B[0])  MulH = np.zeros(M)  MulV = np.zeros(Q)  for i in range(M):  for j in range(N // 2):  MulH[i] += A[i][2 \* j] \* A[i][2 \* j + 1]  for i in range(k1, k2):  for j in range(N // 2):  MulV[i] += B[2 \* j][i] \* B[2 \* j + 1][i]  for i in range(M):  for j in range(k1, k2):  C[i][j] = - MulH[i] - MulV[j]  for k in range(N // 2):  C[i][j] += (A[i][2 \* k] + B[2 \* k + 1][j]) \*  (A[i][2 \* k + 1] + B[2 \* k][j])  if N % 2:  for i in range(M):  for j in range(k1, k2):  C[i][j] = C[i][j] + A[i][N - 1] \* B[N - 1][j] |

Листинг 3. Функция распараллеливания вычислений алгоритма Винограда по столбцам

|  |
| --- |
| def parallel(a, b, f, num):  n = len(a)  m = len(b[0])  c = np.zeros((n, m))  k = n / num  tmp = k  pr\_tmp = 0  threads = []  for i in range(num):  threads.append(threading.Thread(  target=f,  args=(a, b, c, int(pr\_tmp), int(tmp))))  tmp += k  pr\_tmp += k  for i in threads:  i.start()  for i in threads:  i.join()  return c |

# 4. Исследовательская часть

Замеры времени проводились для матриц размера 100х100, 200х200, 300х300, 400х400, 500х500. Так как процессор, на котором выполнялись вычисления, имеет лишь 2 ядра, не было возможности произвести более обширное тестирование такое как, например, на GPU процессоре. Тем не менее, распараллеливание вычислений оказалось эффективнее классического алгоритма, даже учитывая небольшое число ядер процессора.

Очевидно, что производительность алгоритма на матрицах нечетного размера будет хуже, так как внутри алгоритма есть проверка на условие, после чего следует 2 вложенных цикла. Они будут причиной увеличения времени работы. Данное сравнение проводилось в лабораторной работе №2 и в данной работе производиться не будет.

Замер времени проводился с помощью библиотеки time в Python 3.7 и метода process\_time(). На рисунке 1 видно, что во всех случаях попытка распараллелить алгоритм приводила к увеличению производительности алгоритма по времени и сокращению времени работы.

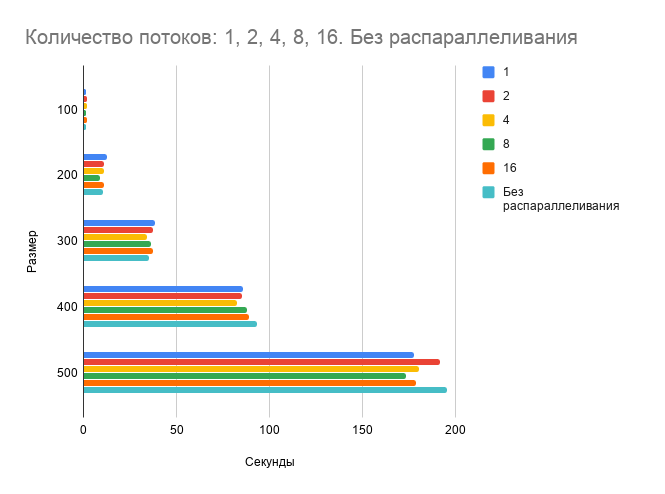


Рисунок 4. Зависимость времени работы алгоритма Винограда от количества потоков

Опыт показал, что в целом рекомендуется применять распараллеливание алгоритма Винограда для увеличения скорости работы.

# Заключение

Была рассмотрена и реализована многопоточная версия алгоритма Винограда, которая показала большую эффективность по сравнению с классическим алгоритмом Винограда. Была оценена трудоёмкость алгоритма Винограда, сделаны схемы алгоритмов. Дана асимптотическая оценка. Задача выполнена.