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 дисциплине "Анализ алгоритмов"

Тема Конвейерная обработка

Студент Шацкий Р.Е.

Группа ИУ7-55Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л.

Москва — 2021 г.

Содержание

Введение

Конвейер[1] — способ организации вычислений, используемый в современных процессорах и контроллерах с целью повышения их производительности (увеличения числа инструкций, выполняемых в единицу времени — эксплуатация параллелизма на уровне инструкций), технология, используемая при разработке компьютеров и других цифровых электронных устройств.

Сам термин «конвейер» пришёл из промышленности, где используется подобный принцип работы — материал автоматически подтягивается по ленте конвейера к рабочему, который осуществляет с ним необходимые действия, следующий за ним рабочий выполняет свои функции над получившейся заготовкой, следующий делает ещё что-то. Таким образом, к концу конвейера цепочка рабочих полностью выполняет все поставленные задачи, сохраняя высокий темп производства. Например, если на самую медленную операцию затрачивается одна минута, то каждая деталь будет сходить с конвейера через одну минуту. В процессорах роль рабочих исполняют функциональные модули, входящие в состав процессора.

Целью данной работы является реализация асинхронного взаимодействия потоков на примере конвейерной обработки данных.

Для достижения поставленной цели необходимо выполнить следующие **задачи:**

1. Изучить асинхронное взаимодействие на примере конвейерной обработки данных.
2. Привести схему конвейерных вычислений.
3. Описать используемые структуры данных.
4. Определить средства программной реализации.
5. Реализовать и протестировать ПО.
6. Провести сравнительный последовательной и конвейерной реализации по затрачиваемым ресурсам (времени работы).
7. Изучить время, затрачиваемое на нахождение заявки в очереди к каждому этапу конвейера.

1 Аналитическая часть

В данном разделе рассматриваются принципы и идея конвейерной обработки данных, а также приводится описание решаемой задачи и выделенных стадий конвейерной обработки.

1.1 Описание конвейерной обработки данных

Конвейер[1] — способ организации вычислений, используемый в современных процессорах и контроллерах с целью повышения их производительности (увеличения числа инструкций, выполняемых в единицу времени — эксплуатация параллелизма на уровне инструкций), технология, используемая при разработке компьютеров и других цифровых электронных устройств.

Идея заключается в параллельном выполнении нескольких инструкций процессора. Сложные инструкции процессора представляются в виде последовательности более простых стадий. Вместо выполнения инструкций последовательно (ожидания завершения конца одной инструкции и перехода к следующей), следующая инструкция может выполняться через несколько стадий выполнения первой инструкции. Это позволяет управляющим цепям процессора получать инструкции со скоростью самой медленной стадии обработки, однако при этом намного быстрее, чем при выполнении эксклюзивной полной обработки каждой инструкции от начала до конца.

Многие современные процессоры управляются тактовым генератором. Процессор внутри состоит из логических элементов и ячеек памяти — триггеров. Когда приходит сигнал от тактового генератора, триггеры приобретают своё новое значение, и «логике» требуется некоторое время для декодирования новых значений. Затем приходит следующий сигнал от тактового генератора, триггеры принимают новые значения, и так далее. Разбивая последовательности логических элементов на более короткие и помещая триггеры между этими короткими последовательностями, уменьшают время, необходимое логике для обработки сигналов. В этом случае длительность одного такта процессора может быть соответственно уменьшена.

1.2 Выделенные стадии конвейерной обработки

В данной работе в качестве алгоритма, реализованного для конвейеризации, используется шифрование паролей с сохранением в базу данных. Таким образом, было выделено три ленты конвейера:

1. Загрузка логинов и паролей из базы данных.

2. Многократное последовательное хеширование полученной строки для большей безопасности.

3. Загрузка полученного значения в базу данных.

1.2.1 Загрузка логинов и паролей из базы данных

Изначальные логины пользователей и пароли для хеширования загружаются из базы данных.

1.2.2 Многократное последовательное хеширование

Существует много алгоритмов хеш-шифрования данных, наиболее распространенные из них приведены ниже [2].

1. MD5 - алгоритм дайджеста сообщений (Message-Digest algorithm), генерирует 128-битное хэш-значение; не может предотвратить коллизии и, следовательно, может быть взломан.

2. SHA-1 - Secure Hash Algorithm 1, может генерировать дайджест сообщения - 160-битное хеш-значение, обычная форма - 40 шестнадцатеричных чисел; не является достаточно безопасным, был обнаружен эффективного метод атаки.

3. SHA-2/SHA-256 - Secure Hash Algorithm 2, генерирует 256-битное хеш-значение; существуют алгоритмы SHA-224, SHA-384, SHA-512, названия которых содержат двоичную длину SHA-2.

4. HMAC - Hash-based Message Authentication Code, использует алгоритм хеширования, принимает сообщение M и ключ K в качестве входных данных и генерирует дайджест сообщения фиксированной длины в качестве выходных данных.

Использование однократного хеширования потенциально небезопасно по нескольким причинам [3]:

- существование коллизий - при одинаковых паролях хеш-значения будут совпадать;
- существование таблиц хэшей для различных функций.

Существует несколько способов увеличения надежности хеш-функций [3]:

1. использование "соли добавление символьной последовательности в конец строки;

2. использование нескольких этапов хеширования, использующих разные хеш-функции;

3. растяжение пароля - итеративный, или рекурсивный, алгоритм, который вычисляет хэш самого себя большое количество раз (на каждом этапе к значению добавляется новая "соль").

В данной работе для обеспечения безопасности хранения данных используется хеш-функция SHA-512 и способ растяжения паролей.

1.2.3 Загрузка в базу данных

По завершении всех предыдущих этапов обработанные пароли пользователей заносятся в локальную реляционную базу данных.

1.3 Требования к программному обеспечению

На основе приведенного алгоритма можно выдвинуть требования к разрабатываемому ПО:

- выходные данные - время работы последовательного выполнения действий (вещественное число);
- время работы конвейера (вещественное число);
- минимальное, среднее, максимальное времена ожидания заявки в очереди каждой из конвейерных лент (вещественные числа);
- наличие обработки некорректного ввода.

1.4 Вывод

Был рассмотрен алгоритм шифрования пароля пользователя и его сохранение в базу данных. Были рассмотрены шаги алгоритма шифрования, выполняемые каждой лентой конвейера для оптимизации работы алгоритма. Выдвинуты требования к разрабатываемому ПО: выходные данные (включающие в себя время работы конвейера, минимальное, среднее, а также максимальное время ожидания заявки в очереди) и наличие обработки некорректного ввода.

2 Конструкторская часть

Данный раздел содержит схемы конвейерной обработки данных, последовательного и конвейерного алгоритма.

2.1 Схемы алгоритмов

В данном пункте раздела представлены схемы реализуемых в работе алгоритмов.

На рисунке 2.1 представлена схема организации конвейерных вычислений на примере конвейера с тремя лентами.

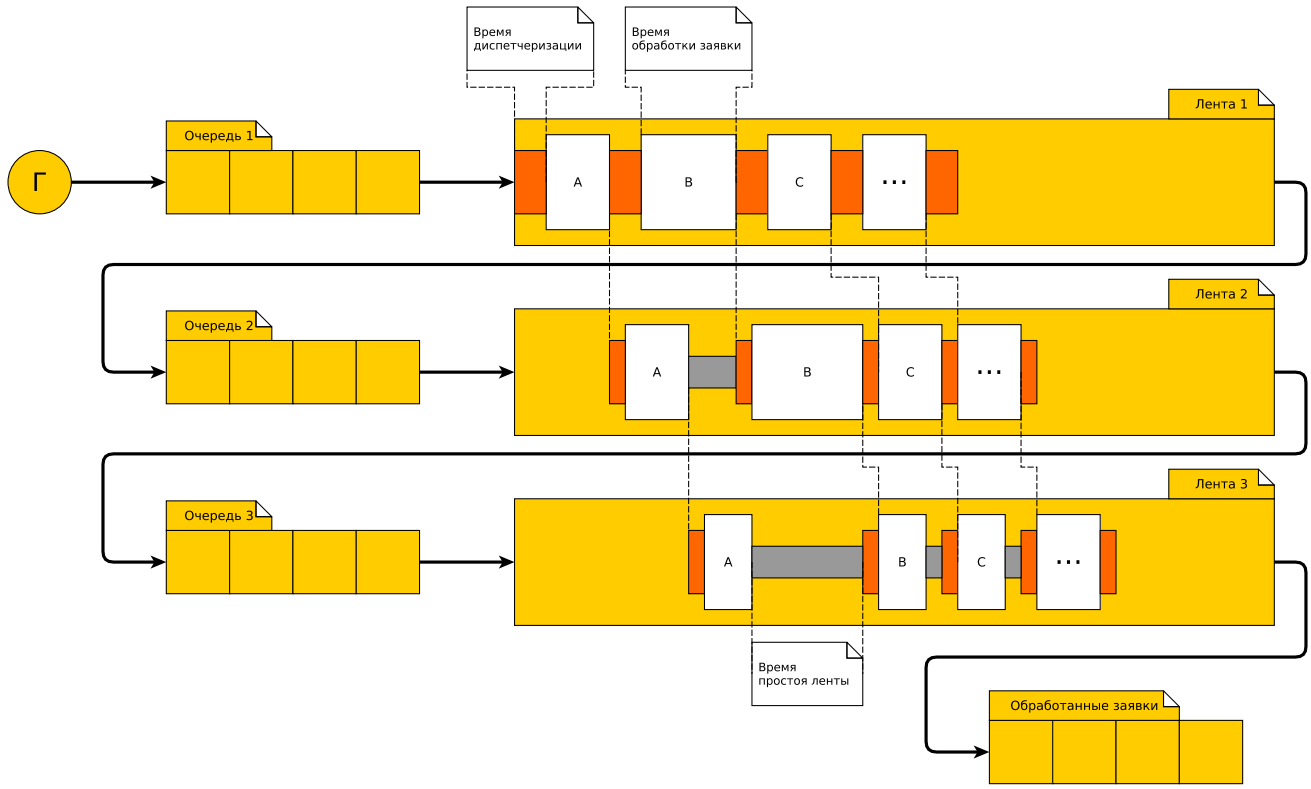

Рис. 2.1: Схема организации конвейера с тремя лентами

2.2 Структуры данных

Для удобства работы были выделены следующие классы:

- UserStats
  - поля
    - times - массив из объектов класса Times, содержащий время поступления и выхода для каждой ленты конвейера;
    - user - объект класса User, содержащий данные пользователя;
    - current - хэш-значение пароля;
    - number - идентификатор объекта класса;
  - методы
    - set_time - установка времени входа/выхода для конкретной ленты;
    - get_time - получение времени входа/выхода для конкретной ленты;
- User
  - поля
    - login - логин пользователя;
    - password - пароль пользователя;
  - методы
    - generate_pass - генерирует случайный пароль;
- Time
  - поля
    - time_in - время поступления на ленту ковейера;
    - time_out - время выхода с ленты конвейера;
  - методы
    - set - установка времени поступления/выхода;
    - get - получение времени поступления/выхода;

Таким образом, программа генерирует заданное в программе количество пользователей и их данных, затем формирует 4 очереди и 3 ленты конвейера и запускает конвейерную обработку. После окончания выводится время конвейерной и последовательной обработки одних и тех же данных, после выводится лог, содержащий время поступления в очередь и выхода задач для каждой ленты конвейера.

2.3 Вывод

В данном разделе приведена схема работы конвейера, выделены структуры данных для дальнейшей реализации программного обеспечения.

3 Технологическая часть

Данный раздел содержит обоснование выбора языка и среды разработки, реализацию алгоритмов.

3.1 Средства реализации

Для реализации программы был выбран язык программирования Python [4]. Такой выбор обусловлен следующими причинами:

- имеется большой опыт разработки;
- имеет большое количество расширений и библиотек, в том числе библиотеку для работы с потоками, измерения времени, построения графиков;
- обладает информативной документацией;

3.2 Реализация алгоритмов

В листингах 3.1 - 3.8 представлены реализации рассматриваемых алгоритмов.

Листинг 3.1: Класс User

```
        PC = 'qwertyuiopasdfghjklzxcvbnm1234567890'
        PCS = len(PC)

        class User:
        def __init__(self, login: str = None, password: str = None):
        if login is None:
        self.__login = str(uuid.uuid4())
        else:
        self.__login = login
        if password is None:
        self.__password = self.__generate_pass()
        else:
        self.__password = password

        @staticmethod
        def __generate_pass():
        size = random.randint(16, 2**9)
        a = ''
        for i in range(size):
        a += PC[random.randint(0, PCS - 1)]
        return a

        @property
        def login(self):
        return self.__login

        @property
        def password(self):
        return self.__password
```

Листинг 3.2: Класс Time

```
1       class Time:
2       def __init__(self):
3       self.__time_in = None
4       self.__time_out = None
5
6       def set(self, t, is_in: bool):
7       if is_in:
8       if self.__time_in is not None:
9       raise Exception("Время поступления на ленту уже записано!")
10      self.__time_in = t
11      elif not is_in:
12      if self.__time_out is not None:
13      raise Exception("Время выхода с ленты уже записано!")
14      self.__time_out = t
15      else:
16      raise Exception("При установке параметр is_in — обязательный!")
17
18      def get(self, is_in: bool = None):
19      if is_in is None:
20      if self.__time_out is None and self.__time_in is None:
21      raise Exception("Время не установлено!")
22      elif self.__time_out is None:
23      raise Exception("Время поступления на ленту не установлено!")
24      elif self.__time_out is None:
25      raise Exception("Время выхода с ленты не установлено!")
26      return [self.__time_in, self.__time_out]
27      elif is_in:
28      if self.__time_out is None:
29      raise Exception("Время поступления на ленту не установлено!")
30      return self.__time_in
31      else:
32      if self.__time_out is None:
33      raise Exception("Время выхода с ленты не установлено!")
34      return self.__time_out
```

Листинг 3.3: Класс UserStats

```
1       class UserStats:
2       cnt = 0
3
4       def __init__(self, login: str = None, password: str = None,
           do_not_init: bool = True):
5       self.__times = [Time() for _ in range(3)]
6       self.__number = self.cnt
7       if do_not_init:
8       self.user = None
9       else:
10      self.user = User(login, password)
11      self.current = ''
12      UserStats.cnt += 1
13
14      def set_time(self, time_, stage: int, is_in: bool):
15      # is_in - True, если на вход, False, если на выход
16      # stage - номер ленты
17      if stage < 0 or stage > 2 or stage is None:
18      raise Exception("Неправильный номер ленты!")
19      self.__times[stage].set(time_, is_in)
20
21      def get_time(self, is_in: bool = None, stage: int = None):
22      # is_in - True, если на вход, False, если на выход, обязательный
23      # stage - номер ленты, None = выдать все
24
25      if stage is not None and (stage < 0 or stage > 2):
26      raise Exception("Неправильный номер ленты!")
27      if stage is None:
28      if is_in is None:
29      return [self.__times[i].get() for i in range(3)]
30      else:
31      return [self.__times[i].get(is_in) for i in range(3)]
32      else:
33      return self.__times[stage].get(is_in)
34
35      def get_number(self):
36      return self.__number
```

Листинг 3.4: Файл master.py (часть 1)

```
1         def init_db(con: sqlite3.Connection):
2         cur = con.cursor()
3         cur.execute('create table if not exists users_out (login text,
              password text)')
4         cur.execute('create table if not exists users_in (i int, login text,
              password text)')
5         con.commit()
6
7         def fill_input_db(con: sqlite3.Connection, count=30):
8         cur = con.cursor()
9         for index in range(count):
10        m = User()
11        cur.execute(f'insert into users_in (i, login, password) values ({index
              }, "{m.login()}", "{m.password()}")')
12        con.commit()
13
14        def get_list_size_from_db(con: sqlite3.Connection) -> int:
15        cur = con.cursor()
16        cur.execute('select count(*) from users_in')
17        return cur.fetchone()[0]
18
19        def clear_db(con: sqlite3.Connection):
20        cur = con.cursor()
21        cur.execute('drop table if exists users_out')
22        cur.execute('drop table if exists users_in')
23        con.commit()
24
25        def generate_users(users_count: int) -> [UserStats]:
26        users = []
27
28        for i in range(users_count):
29        users.append(UserStats(do_not_init=False))
30
31        return users
```

Листинг 3.5: Файл master.py (часть 2)

```
    def load_user(u: UserStats):
    con = sqlite3.connect('app.db')
    c = con.cursor()
    c.execute(f'select login, password from users_in where i = {u.
        get_number()}')
    login, password = c.fetchone()
    u.user = User(login, password)

    def get_hashed(u: UserStats):
    u.current = u.user.login() + 'my secret key'

    for iter in range(2000):
    u.current = sha512((u.current + str(iter)).encode('Utf-8')).hexdigest
        ()

    def insert(u: UserStats):
    con = sqlite3.connect('app.db')
    c = con.cursor()
    c.execute(f"insert into users_out (login, password) values ('{u.user.
        login()}', '{u.current}')")
    con.commit()
    con.close()

    def tex_table(stats, stages):
    print("\\hline")
    print('Stage N & Task M & Start Time & End Time\\\\')
    print("\\hline")

    for stat_num, stat in enumerate(stats):
    for stage in range(stages):
    times = stat.get_time(stage=stage)
    print(
    f'Stage: {stage + 1} & Task: {stat_num + 1} & {times[0] - start_time
        :.6f} & {times[1] - start_time:.6f} \\\\')

    print("\\hline")
```

Листинг 3.6: Файл master.py (часть 3)

```
    def job(task: Callable, in_queue: SimpleQueue, out_queue: SimpleQueue,
        stage: int):
    while True:
    data: UserStats = in_queue.get()

    if data is None:
    out_queue.put(data)
    break

    data.set_time(time.time(), stage, True)
    task(data)
    data.set_time(time.time(), stage, False)
    out_queue.put(data)

    def test_serial(users):
    for user in users:
    load_user(user)
    get_hashed(user)
    insert(user)

    if __name__ == '__main__':
    stages_count = 3

    connection = sqlite3.connect('app.db')
    clear_db(connection)
    init_db(connection)
    fill_input_db(connection)

    _users = generate_users(get_list_size_from_db(connection))

    pipeline_time = 0
    serial_time = 0
    cnt = 10
    for i in range(cnt):
    clear_db(connection)
    init_db(connection)
    fill_input_db(connection)

    passwords_queue = SimpleQueue()
    salt_queue = SimpleQueue()
```

Листинг 3.7: Файл master.py (часть 4)

```
    hash_queue = SimpleQueue()
    result_queue = SimpleQueue()

    add_salter = Process(target=job, args=(load_user, passwords_queue,
        salt_queue, 0))
    hasher = Process(target=job, args=(get_hashed, salt_queue, hash_queue,
```

```
        1))
    inserter = Process(target=job, args=(insert, hash_queue, result_queue,
        2))
    pipeline = [add_salter, hasher, inserter]

    for u in _users:
    passwords_queue.put(u)

    passwords_queue.put(None)
    start_time = time.time()
    for worker in pipeline:
    worker.start()

    for worker in pipeline:
    worker.join()
    end_time = time.time()
    pipeline_time += end_time - start_time

    start_time_ = time.time()
    test_serial(_users)
    end_time_ = time.time()
    serial_time += end_time_ - start_time_

    pipeline_time /= cnt

    print(f'Pipeline time = {pipeline_time * 1e6} mks')

    serial_time /= cnt
    print(f'Serial time = {serial_time * 1e6} mks')
    print('Press to get log')
    input()

    stats = []
```

Листинг 3.8: Файл master.py (часть 5)

```
        while not result_queue.empty():
        stats.append(result_queue.get())
        stats.pop()

        deltas = [[], [], []]
        for stat in stats:
        for stage in range(stages_count):
        stage_stat = stat.get_time(stage=stage)
        deltas[stage].append(stage_stat[1] - stage_stat[0])

        for stage in range(stages_count):
        print(f'Max time on stage {stage + 1} = {max(deltas[stage]) * 1e6} mks
            ')
        print(f'Min time on stage {stage + 1} = {min(deltas[stage]) * 1e6} mks
            ')
        print(f'Avg time on stage {stage + 1} = {sum(deltas[stage]) / len(
            deltas[stage]) * 1e6} mks\n')

        connection.close()

        tex_table(stats, stages_count)
```

3.3 Выводы

В данном разделе была реализована конвейерная обработка данных: изменение, хеширование и сохранение паролей пользователей.

4 Экспериментальная часть

В данном разделе сравниваются реализованные алгоритмы, дается сравнительная оценка затрат на время.

4.1 Пример работы программы

Пример работы программы представлен на рисунке 4.1.

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование:

- операционная система: Ubuntu 20.01 Linux x86_64 [5];
- оперативная память: 8 Гб;
- процессор: AMD Ryzen5 4500U [6]:
  - количество физических ядер: 6;
  - количество логических ядер: 6.

```
Pipeline time = 133182.73544311523 mks
Serial time = 228827.65769958496 mks
Press to get log

Max time on stage 1 = 812.7689361572266 mks
Min time on stage 1 = 133.99124145507812 mks
Avg time on stage 1 = 244.4903055826823 mks

Max time on stage 2 = 12578.487396240234 mks
Min time on stage 2 = 3556.489944458008 mks
Avg time on stage 2 = 9844.144185384115 mks

Max time on stage 3 = 3536.4627838134766 mks
Min time on stage 3 = 2316.9517517089844 mks
Avg time on stage 3 = 3023.9184697469077 mks

\hline
Stage N & Task M & Start Time & End Time\\
\hline
Stage: 1 & Task: 1 & 0.002179 & 0.002991 \\
Stage: 2 & Task: 1 & 0.003429 & 0.016008 \\
Stage: 3 & Task: 1 & 0.016438 & 0.019504 \\
\hline
Stage: 1 & Task: 2 & 0.003314 & 0.003629 \\
Stage: 2 & Task: 2 & 0.016270 & 0.026802 \\
Stage: 3 & Task: 2 & 0.027042 & 0.030579 \\
\hline
Stage: 1 & Task: 3 & 0.003776 & 0.004056 \\
Stage: 2 & Task: 3 & 0.026944 & 0.037537 \\
Stage: 3 & Task: 3 & 0.037839 & 0.040872 \\
\hline
Stage: 1 & Task: 4 & 0.004245 & 0.004612 \\
Stage: 2 & Task: 4 & 0.037680 & 0.048180 \\
Stage: 3 & Task: 4 & 0.048373 & 0.051491 \\
\hline
Stage: 1 & Task: 5 & 0.004766 & 0.005046 \\
Stage: 2 & Task: 5 & 0.048321 & 0.058671 \\
Stage: 3 & Task: 5 & 0.058857 & 0.062011 \\
\hline
Stage: 1 & Task: 6 & 0.005198 & 0.005489 \\
Stage: 2 & Task: 6 & 0.058783 & 0.069260 \\
Stage: 3 & Task: 6 & 0.069451 & 0.071884 \\
\hline
Stage: 1 & Task: 7 & 0.005622 & 0.005898 \\
Stage: 2 & Task: 7 & 0.069384 & 0.079809 \\
Stage: 3 & Task: 7 & 0.080004 & 0.083281 \\
\hline
Stage: 1 & Task: 8 & 0.006060 & 0.006336 \\
Stage: 2 & Task: 8 & 0.079923 & 0.090405 \\
Stage: 3 & Task: 8 & 0.090638 & 0.093818 \\
\hline
Stage: 1 & Task: 9 & 0.006461 & 0.006736 \\
Stage: 2 & Task: 9 & 0.090529 & 0.094146 \\
Stage: 3 & Task: 9 & 0.094260 & 0.096577 \\
\hline
```

Рис. 4.1: Пример работы программы

4.3 Время выполнения алгоритмов

Время выполнения алгоритмов измерялось на автоматически генерируемых в необходимом количестве пользовательских данных с использованием функции time библиотеки time. Усредненные результаты 10 замеров реального времени работы приведены в таблице ниже.

На рисунке 4.2 представлена зависимость времени выполнения алгоритма в зависимости от "плана на день количества пользователей для обработки - на основе таблицы 4.1. Последовательное выполнение занимает в среднем в 1.5-1.7 раз больше времени, чем конвейерное с незначительным увеличением выигрыша обработки на конвейере при увеличении количества пользователей. Таким образом, выигрыш от использования конвейерной обработки при количестве пользователей от 15 до 85 приблизительно одинаков и составляет около 1.6 раз.

Таблица 4.1: Время выполнения алгоритма для разного количества пользователей в микросекундах

Размер	Последовательное выполнение	Конвейерная обработка
5	38291.287	36348.152
15	109756.374	73250.5798
25	181147.265	113172.388
35	252654.957	152421.474
45	322310.686	197659.373
55	399007.916	239235.663
65	461043.334	286625.170
75	541337.20	321961.14

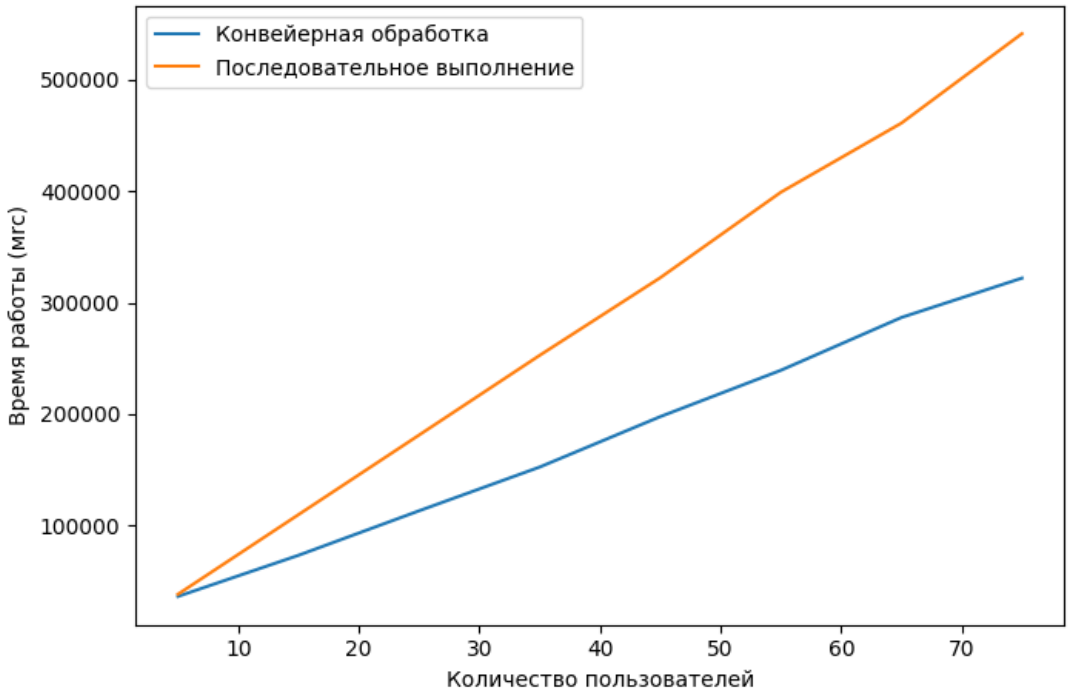

Рис. 4.2: Зависимость времени выполнения от количества пользователей

4.4 Тестирование

Для тестирования корректности работы ПО используется анализ логов. Полученная для последних 6 заявок таблица приведена ниже. На рисунке 4.3 представлены данные о максимальном, минимальном и среднем времени обработки заявок на каждой ленте конвейера.

Таблица 4.2: Полученный лог

Stage N	Task M	Start Time	End Time
Stage: 1	Task: 24	0.012759	0.012983
Stage: 2	Task: 24	0.096859	0.100533
Stage: 3	Task: 24	0.100766	0.103546
Stage: 1	Task: 25	0.013081	0.013289
Stage: 2	Task: 25	0.100624	0.104318
Stage: 3	Task: 25	0.104567	0.107468
Stage: 1	Task: 26	0.013381	0.013590
Stage: 2	Task: 26	0.104418	0.108121
Stage: 3	Task: 26	0.108363	0.111242
Stage: 1	Task: 27	0.013669	0.017236
Stage: 2	Task: 27	0.108238	0.111867
Stage: 3	Task: 27	0.112061	0.114830
Stage: 1	Task: 28	0.017447	0.017701
Stage: 2	Task: 28	0.111984	0.115609
Stage: 3	Task: 28	0.115740	0.118498
Stage: 1	Task: 29	0.017766	0.017931
Stage: 2	Task: 29	0.115671	0.119328
Stage: 3	Task: 29	0.119419	0.122284
Stage: 1	Task: 30	0.017987	0.018134
Stage: 2	Task: 30	0.119397	0.123048
Stage: 3	Task: 30	0.123159	0.126240

```
Max time on stage 1 = 3566.265106201172 mks
Min time on stage 1 = 146.38900756835938 mks
Avg time on stage 1 = 374.62711334228516 mks

Max time on stage 2 = 6560.087203979492 mks
Min time on stage 2 = 3587.484359741211 mks
Avg time on stage 2 = 3821.8100865681968 mks

Max time on stage 3 = 4501.3427734375 mks
Min time on stage 3 = 2593.517303466797 mks
Avg time on stage 3 = 2849.3324915568037 mks
```

Рис. 4.3: Время выполнения заявок на каждой из лент конвейера

4.5 Выводы

В данном разделе были проведены измерения времени, затрачиваемого на загрузку данных о пользователе из базы данных, хеширование пароля и сохранения в базу данных полученного хэша. Для размера входной очереди конвейера, принадлежащего интервалу [5, 85],

выигрыш по сравнению с последовательной обработкой составил приблизительно 1.5-1.7 раз с небольшими колебаниями при увеличении количества пользователей.

Заключение

В процессе выполнения лабораторной работы было реализовано асинхронное взаимодействие потоков на примере конвейерной обработки данных. В качестве конвейера рассмотрена работа с логинами и паролями пользователей: загрузка из базы данных, многократное хеширование хеш-функцией SHA-512 с добавлением "солей"и сохранение полученных данных в базу данных.

Было исследовано время выполнения выше обозначенного алгоритма. В результате было выявлено, что в среднем для количества пользователей от 5 до 85 выигрыш от использования конвейерной обработки составляет 1.6 раз с небольшим увеличением при увеличении количества пользователей.

Литература

[1] Параллельная обработка данных. Режим доступа: https://parallel.ru/vvv/lec1.html. Дата обращения: 21.11.2021.

[2] Подробное объяснение хеш-шифрования и инструментов md5, sha1, sha256, Java. Режим доступа: https://russianblogs.com/article/80151483863/. Дата обращения: 21.11.2021.

[3] Риски и проблемы хеширования паролей. Режим доступа: https://itnan.ru/post.php?c=1p=271245. Дата обращения: 21.11.2021.

[4] About Python [Электронный ресурс]. Режим доступа: https://www.python.org/about/. Дата обращения: 20.10.2021.

[5] Ubuntu по-русски [Электронный ресурс]. Режим доступа: https://ubuntu.ru/. Дата обращения: 20.10.2021.

[6] AMD Ryzen™ 5 4500U. Режим доступа: https://www.amd.com/ru/products/apu/amd-ryzen-5-4500u. Дата обращения: 20.10.2021.