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»

Отчёт по лабораторной работе №5 по курсу "Анализ алгоритмов"

Тема Конвейерные вычисления

Студент Леонов В.В.

Группа ИУ7-56Б

Оценка (баллы) _____

Преподаватель Волкова Л.Л.

Москва — 2021 г.

Содержание

Введение

21 век можно по праву назвать веком многоядерных машин. Многоядерность позволяет существенным образом повысить быстродействие систем. Существует множество подходов для использования многоядерности для ускорения вычислений. В ситуациях, когда один и тот же набор данных необходимо обработать несколькими независимыми алгоритмами, используют конвейерный подход по аналогии с традиционным подходом производства на фабриках. На каждый этап обработки данных выделяется отдельная конвейерная лента, которая отвечает за конкретную подзадачу [1].

Целью данной лабораторной работы являются изучение и реализация конвейерных вычислений.

Для достижения указанной выше цели следует выполнить следующие задачи:

- рассмотреть и изучить асинхронную конвейерную обработку данных;
- сформулировать задачу, которая будет реализованна на конвейере;
- определить требования к ПО;
- построить схемы алгоритмов, решающих поставленную задачу с использованием последовательного и параллельного конвейера;
- описать структуру ПО;
- исходя из полученных эксперементально данных, провести сравнительный анализ алгоритмов, реализованных линейно и параллельно;
- описание и обоснование полученных результатов в отчете о выполненной лабораторной работе.

1 Аналитическая часть

1.1 Конвейерная обработка данных

Пусть операция разбита на микрооперации. Расположим микрооперации в порядке выполнения и для каждого выполнения выделим отдельную часть устройства. В первый момент времени входные данные поступают для обработки в первую часть. После выполнения первой микрооперации первая часть передает результаты своей работы второй части, а сама берет новые данные. Когда входные аргументы пройдут все этапы обработки, на выходе устройства появится результат выполнения операции. Таким образом, реализуется функциональный параллелизм. Каждая часть устройства называется ступенью конвейера, а общее число ступеней – длиной конвейера.

Каждая из ступеней конвейера использует отдельный поток и очередь для организации корректной работы системы.

1.2 Описание предметной области

В качестве алгоритма, реализованного для распределения на конвейере, был выбран абстрактный процесс генерации дневного рациона пищи для клиентов службы доставки еды:

- процесс подбора блюда на завтрак;
- процесс подбора блюда на обед;
- процесс подбора блюда на ужин.

1.3 Вывод

Для ПО, решающего поставленную задачу следует выделить следующие требования:

- ПО должно генерировать корректное меню дневного рациона;
- ПО должно генерировать один дневной рацион не более чем за 50 мс;
- ПО должно обеспечить методы контроля времени обработки каждого из этапов конвейера.

Были рассмотрены принципы и особенности построения конвейерных вычислений, а также подобрана задача для моделирования поставленной задачи.

2 Конструкторская часть

В данном разделе представлены схемы алгоритмов, описанных в аналитическом разделе.

2.1 Схемы алгоритмов

На рисунке 2.1 приведена схема линейного алгоритма.

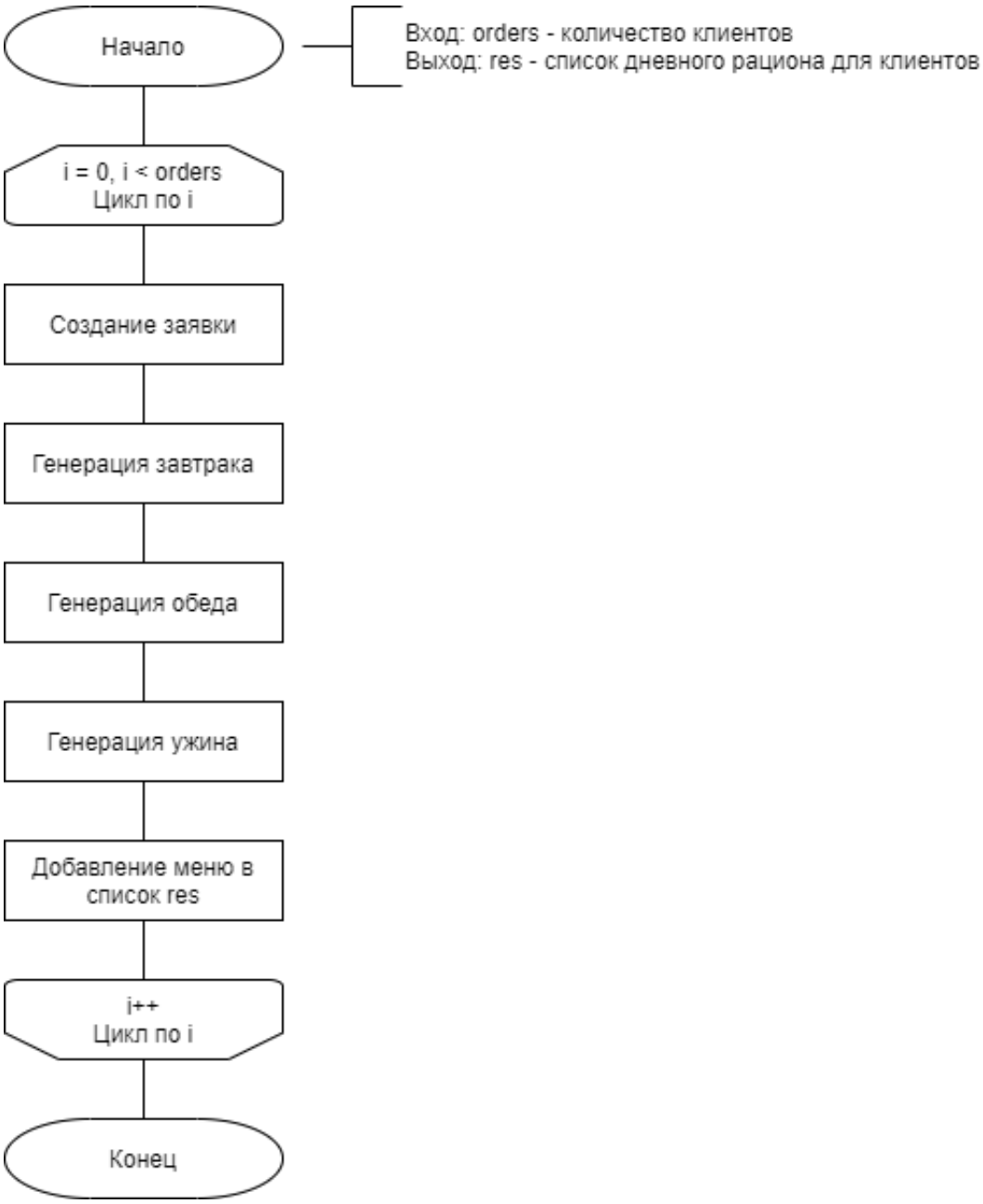

Рисунок 2.1 – Схема линейного алгоритма

На рисунке 2.2 приведена схема многопоточного конвейерного алгоритма.

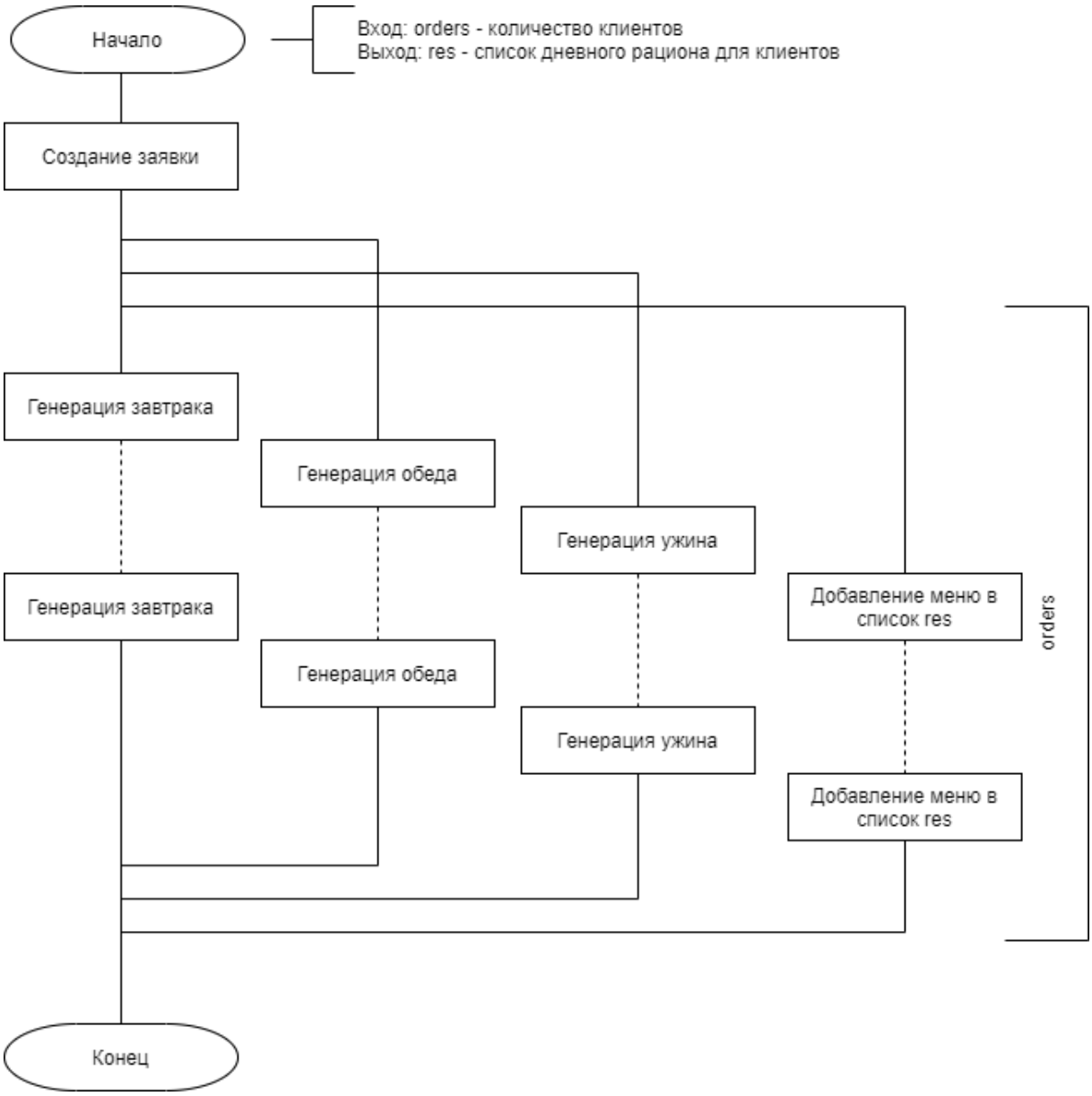

Рисунок 2.2 – Схема многопоточного конвейерного алгоритма

2.2 Описание структуры ПО

Программа будет будет включать в себя один смысловой модуль называемый *conveyor*, который будет содержать в себе несколько файлов реализующих процедуры и функции, связанные с реализацией параллельного и синхронного конвейера. Независимо от модуля будет существовать файл

main.go, реализующий взаимодействие между пользователем и модулем, описанным ранее.

2.3 Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы требуемых алгоритмов.

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подается количество заявок на создание дневного рациона;
- на выходе лог обработки заявок с временными метками;
- существует система автоматического тестирования.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран многопоточный язык GO [2]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного языка. Также данный язык предоставляет удобные средства для работы с потоками (goroutines) [3].

Время работы алгоритмов было замерено с помощью функции `Now()` из библиотеки `Time`.

3.3 Листинг кода

В листингах 3.1–?? приведены линейная и конвейерная реализации алгоритма, а также вспомогательные функции.

Листинг 3.1 – Линейный алгоритм

```
func Linear(orders int) []*Ration_t {
    var secondLineQ, thirdLineQ Queue
    var result []*Ration_t

    for {
        if len(result) != orders {
            ration := new(Ration_t)

            ration.startBreakfast = time.Now()
            ration.breakfast = genBreakfast()
            ration.finishBreakfast = time.Now()

            secondLineQ.push(ration)

            if !secondLineQ.isEmpty() {
                ration = secondLineQ.pop()

                ration.startLunch = time.Now()
                ration.lunch = genLunch()
                ration.finishLunch = time.Now()

                thirdLineQ.push(ration)
            }

            if !thirdLineQ.isEmpty() {
                ration = thirdLineQ.pop()

                ration.startDinner = time.Now()
                ration.dinner = genDinner()
                ration.finishDinner = time.Now()

                result = append(result, ration)
            }
        } else {
            return result
        }
    }
}
```

Листинг 3.2 – Конвейерный алгоритм

```
func Parallel(orders int) []*Ration_t {
    firstLine := make(chan *Ration_t, orders)
    secondLine := make(chan *Ration_t, orders)
    thirdLine := make(chan *Ration_t, orders)
    result := make([]*Ration_t, 0, orders)
    var wg sync.WaitGroup
    wg.Add(3)
    breakfast := func() {
        for ration := range firstLine {
            ration.startBreakfast = time.Now()
            ration.breakfast = genBreakfast()
            ration.finishBreakfast = time.Now()
            secondLine <- ration
        }
    }
    lunch := func() {
        for ration := range secondLine {
            ration.startLunch = time.Now()
            ration.lunch = genLunch()
            ration.finishLunch = time.Now()
            thirdLine <- ration
        }
    }
    dinner := func() {
        for ration := range thirdLine {
            ration.startDinner = time.Now()
            ration.dinner = genDinner()
            ration.finishDinner = time.Now()
            result = append(result, ration)

            if len(result) == cap(result) {
                wg.Done()
                wg.Done()
                wg.Done()
            }
        }
    }
    go breakfast()
    go lunch()
    go dinner()
    for i := 0; i < orders; i++ {
        ration := new(Ration_t)
        firstLine <- ration
    }
    wg.Wait()
    return result
}
```

Листинг 3.3 – Алгоритмы логирования

```
)

func LogTime(orders []*Ration_t) {
    timestamp := orders[0].startBreakfast

    fmt.Printf("Stage␣Starting␣Time\n")
    for id, order := range orders {
        fmt.Printf("Order␣ID␣=␣%3v,␣1␣Stage␣=␣%v,␣2␣Stage␣=␣%v,␣3␣Stage␣=␣%v\n",
            id,
            order.startBreakfast.Sub(timestamp),
            order.startLunch.Sub(timestamp),
            order.startDinner.Sub(timestamp))
    }

    fmt.Printf("Stage␣Finishing␣Time\n")
    for id, order := range orders {
        fmt.Printf("Order␣ID␣=␣%3v,␣1␣Stage␣=␣%v,␣2␣Stage␣=␣%v,␣3␣Stage␣=␣%v\n",
            id,
            order.finishBreakfast.Sub(timestamp),
            order.finishLunch.Sub(timestamp),
            order.finishDinner.Sub(timestamp))
    }

    var fDowntime, sDowntime, tDowntime time.Duration
    for i := 1; i < len(orders); i++ {
        fDowntime += orders[i].startBreakfast.Sub(timestamp) -
            orders[i-1].finishBreakfast.Sub(timestamp)
        sDowntime += orders[i].startLunch.Sub(timestamp) -
            orders[i-1].finishLunch.Sub(timestamp)
        tDowntime += orders[i].startDinner.Sub(timestamp) -
            orders[i-1].finishDinner.Sub(timestamp)
    }
    fmt.Printf("Downtimes\n")
    fmt.Printf("␣␣␣␣1␣Stage␣=␣%v,␣2␣Stage␣=␣%v,␣3␣Stage␣=␣%v\n",
        fDowntime, sDowntime, tDowntime)
}

func LogTimeLatex(orders []*Ration_t, filename string) {
    timestamp := orders[0].startBreakfast

    file, err := os.Create("./" + filename + ".csv")
    if err != nil {
```

Листинг 3.4 – Используемые структуры данных

```
type Ration_t struct {
    startBreakfast time.Time
    finishBreakfast time.Time
    startLunch time.Time
    finishLunch time.Time
    startDinner time.Time
    finishDinner time.Time

    breakfast Breakfast_t
    lunch Lunch_t
    dinner Dinner_t
}

type Breakfast_t struct {
    maincourse string
    fruit string
}

type Lunch_t struct {
    maincourse string
    snack string
}

type Dinner_t struct {
    maincourse string
    dessert string
}
```

Листинг 3.5 – Алгоритмы обработки очереди

```
func (queue *Queue) push(ration *Ration_t) {
    *queue = append(*queue, ration)
}

func (queue *Queue) pop() *Ration_t {
    var ration *Ration_t
    *queue, ration = (*queue)[1:], (*queue)[0]
    return ration
}

func (queue *Queue) isEmpty() bool {
    return len(*queue) == 0
}
```

3.4 Вывод

Были разработаны линейный и конвейерный спроектированные алгоритмы генерации дневного рациона клиента службы доставки.

4 Исследовательская часть

4.1 Пример работы

Демонстрация работы программы приведена на рисунке 4.1.

```
Enter the amount of orders:5
Parallel algorithm:
Stage Starting Time
Order ID =   0, 1 Stage = 0s, 2 Stage = 11.170519ms, 3 Stage = 23.072936ms
Order ID =   1, 1 Stage = 11.063765ms, 2 Stage = 22.926367ms, 3 Stage = 40.593727ms
Order ID =   2, 1 Stage = 17.705443ms, 2 Stage = 36.44421ms, 3 Stage = 50.677995ms
Order ID =   3, 1 Stage = 33.451793ms, 2 Stage = 50.633456ms, 3 Stage = 69.357885ms
Order ID =   4, 1 Stage = 48.285906ms, 2 Stage = 69.280597ms, 3 Stage = 84.048969ms
Stage Finishing Time
Order ID =   0, 1 Stage = 11.020182ms, 2 Stage = 22.876672ms, 3 Stage = 40.589849ms
Order ID =   1, 1 Stage = 17.699742ms, 2 Stage = 36.439551ms, 3 Stage = 49.301997ms
Order ID =   2, 1 Stage = 33.44474ms, 2 Stage = 50.625921ms, 3 Stage = 57.092615ms
Order ID =   3, 1 Stage = 48.280045ms, 2 Stage = 69.274085ms, 3 Stage = 84.044841ms
Order ID =   4, 1 Stage = 67.761125ms, 2 Stage = 80.914832ms, 3 Stage = 98.670952ms
Downtimes
    1 Stage = 62.198µs, 2 Stage = 68.401µs, 3 Stage = 13.649274ms
Linear algorithm:
Stage Starting Time
Order ID =   0, 1 Stage = 0s, 2 Stage = 2.312653ms, 3 Stage = 5.964013ms
Order ID =   1, 1 Stage = 7.266201ms, 2 Stage = 9.648326ms, 3 Stage = 14.053907ms
Order ID =   2, 1 Stage = 19.378629ms, 2 Stage = 19.385101ms, 3 Stage = 29.328607ms
Order ID =   3, 1 Stage = 38.303371ms, 2 Stage = 49.864577ms, 3 Stage = 63.46126ms
Order ID =   4, 1 Stage = 66.907519ms, 2 Stage = 66.917077ms, 3 Stage = 84.675306ms
Stage Finishing Time
Order ID =   0, 1 Stage = 2.310348ms, 2 Stage = 5.960847ms, 3 Stage = 7.216095ms
Order ID =   1, 1 Stage = 9.645691ms, 2 Stage = 14.050691ms, 3 Stage = 19.373159ms
Order ID =   2, 1 Stage = 19.38438ms, 2 Stage = 29.325931ms, 3 Stage = 38.297259ms
Order ID =   3, 1 Stage = 49.861791ms, 2 Stage = 63.459056ms, 3 Stage = 66.901247ms
Order ID =   4, 1 Stage = 66.914943ms, 2 Stage = 84.672821ms, 3 Stage = 84.680245ms
Downtimes
    1 Stage = 50.65351ms, 2 Stage = 33.018556ms, 3 Stage = 59.73132ms
```

Рисунок 4.1 – Демонстрация работы алгоритмов сортировок

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: Windows 10 64-bit [4];

- память: 16 GB;
- процессор: AMD Ryzen 5 4600H [5] @ 3.00 GHz.

Тестирование проводилось на ноутбуке при включённом режиме производительности. Во время тестирования ноутбук был нагружен только системными процессами.

4.3 Время выполнения алгоритмов

В таблицах 4.1-4.2 приведены временные отметки работы линейного и параллельного конвейера.

Таблица 4.1 – Лог обработки заявок на линейном конвейере

№ заявки	№ этапа	Время начала	Время конца
1	1	0s	11.310012ms
1	2	11.311935ms	13.47765ms
1	3	13.478612ms	22.957808ms
2	1	22.960773ms	28.122197ms
2	2	28.123219ms	34.346058ms
2	3	34.34691ms	34.349064ms
3	1	34.350367ms	50.734355ms
3	2	50.735587ms	61.251483ms
3	3	61.252795ms	66.435439ms
4	1	66.438625ms	81.97257ms
4	2	81.974224ms	85.301686ms
4	3	85.303239ms	88.611855ms
5	1	88.633807ms	96.07894ms
5	2	96.080503ms	104.570381ms
5	3	104.571714ms	114.104312ms

Таблица 4.2 – Лог обработки заявок на параллельном конвейере

№ заявки	№ этапа	Время начала	Время конца
1	1	0s	19.372072ms
1	2	19.447747ms	34.107102ms
1	3	34.169231ms	50.547227ms
2	1	19.393663ms	21.755132ms
2	2	34.165313ms	50.508273ms
2	3	50.54841ms	56.741513ms
3	1	21.756845ms	23.112743ms
3	2	50.510327ms	67.960691ms
3	3	68.045643ms	75.5349ms
4	1	23.114296ms	42.775641ms
4	2	67.964217ms	71.204754ms
4	3	75.537375ms	79.846261ms
5	1	42.779418ms	45.396334ms
5	2	71.206617ms	72.406458ms
5	3	79.848515ms	97.096062ms

На рисунке 4.2 приведен график зависимости времени работы алгоритмов для различных наборов данных.

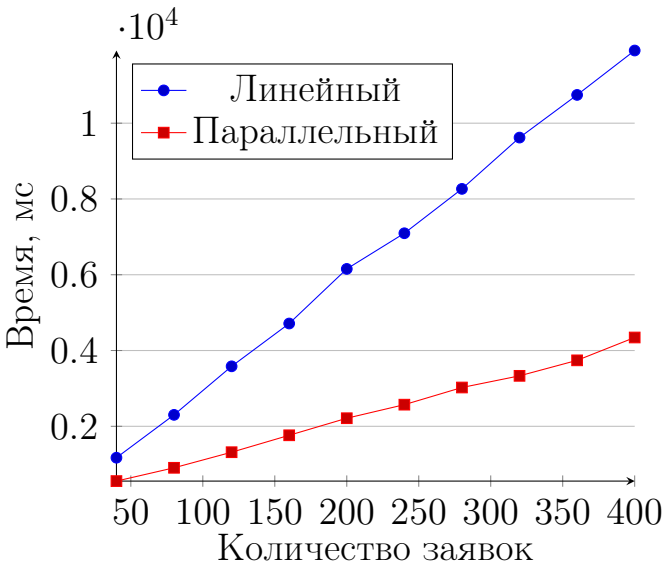

Рисунок 4.2 – Сравнение скорости работы алгоритмов конвейера при различном количестве заявок.

4.4 Вывод

Исходя из полученных эксперементальных данных можно сделать следующие выводы:

- реализация многопоточного алгоритма на основе конвейейрного подхода позволяет получить выигрыш по быстродействию порядка 2.87 раз (при условии, что рассматриваемый алгоритм состоит из трех независимых алгоритмов, каждый из которых обрабатывается на отдельном потоке);

- при реализации многопоточного алгоритма наблюдается многократное уменьшение времени простоя каждой из лент конвейера по сравнению с его линейным вариантом (приблизительно в 1000 раз).

Заключение

В ходе выполнения работы были выполнены все поставленные задачи и изучены методы динамического многопоточного программирования на основе алгоритма генерации дневного рациона для клентов службы доставки еды.

Реализация многопоточного алгоритма на основе конвейейрного подхода позволяет получить выигрыш по быстродействию порядка 2.87 раз (при условии, что рассматриваемый алгоритм состоит из трех независимых алгоритмов, каждый из которых обрабатывается на отдельном потоке).

При реализации многопоточного алгоритма наблюдается многократное уменьшение времени простоя каждой из лент конвейера по сравнению с его линейным вариантом (приблизительно в 1000 раз).

Параллельные конвейерные вычисления позволяют организовать непрерывную обработку данных, что позволяет выиграть время в задачах, где требуется обработка больших объемов информации за небольшой промежуток времени.

Список литературы

[1] Пыхтин А.А. Ушаков С.В. Параллельные конвейерные вычисления. – Курск, КГУ, 2016.

[2] The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/` (дата обращения: 01.10.2021).

[3] Neskorod J. Go Concurrency. – MIT, 2020.

[4] Explore Windows 10. [Электронный ресурс]. Режим доступа: `https://www.microsoft.com/en-us/windows/` (дата обращения: 02.10.2021).

[5] AMD Processors [Электронный ресурс]. Режим доступа: `https://www.amd.com/en/products/apu/amd-ryzen-5-4600h` (дата обращения: 02.10.2021).