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»
ДИСЦИПЛИНА «Анализ алгоритмов»

Рубежный контроль №1

Дисциплина Анализ алгоритмов
Тема Стена Фокса
Студент Боренко А. Д.
Группа ИУ7-52Б
Оценка (баллы) _____
Преподаватель Волкова Л.Л.

Москва — 2022г.

Содержание

Введение

Параллельные вычисления — способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно). Такие вычисления обычно реализуются на вычислительных системах, состоящих из множества вычислительных узлов, что значительно увеличивает скорость выполнения задачи. [1]

В случае, если общую задачу можно разделить на независимые маленькие подзадачи, то параллельные потоки будут работать эффективно все время. Однако, если последующие подзадачи зависят от предыдущих, то потоки могут тормозить друг друга. Особенно это заметно, если один из потоков медленнее других.

Рассморим задачу построения кирпичной стены - стены Фокса. Новый кирпич из ряда выше первого можно положить, только если на местах под ним уже лежат 2. Предположим, что в системе 4 вычислительных узла - каменщика. Один из них медленее других. Тогда 3 каменщика выложат все кирпичи, которые смогут, из своей области, и будут ждать пока четвертый выложит свою часть. Та же проблема возникнет, если производительность каменщиков абсолютно одинакова, но кирпичи (подзадачи) разные по сложности. Таким образом, стена будет строится со скоростью наиболее медленного каменщика.

Этот показатель можно улучшить, если распланировать нагрузку заранее, так чтобы потоки не простаивали. Распланировать заранее можно, если известны характеристики потоков и задач.

Однако такие данные о задачах и процессах известны не всегда. Тогда можно планировать нагрузку можно в процессе работы, добавив в систему поток-диспетчер, который будет просыпаться через определенные заранее промежутки времени, проверять как справляются потоки, и регулировать границы их ответственности (их нагрузку), в соответствии с результатами их работы. Такое решение задачи называют динамической балансировкой. [2]

Целью данной лабороторной работы является исследование динамической ба-

лансировки на примере построения кирпичной стены. Задачами данной лабораторной являются:

1. реализация алгоритма построения кирпичной стены с балансировкой;
2. визуализация работы исслеуемого алгоритма;
3. описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчетно-пояснительная записка к работе.

1. Аналитическая часть

В данном разделе будет рассмотрен алгоритм построения кирпичной стены с динамической балансировкой нагрузки.

1.1 Алгоритм построения кирпичной стены с диспетчером

Построение стены - это укладывани кирпичей в шахматном порядке. Кирпич не может быть уложен в "воздух т.е. под кирпичем должна быть, либо земля, либо два других кирпича. Каждый каменщик строит свой участок стены, и отчитывается о том, сколько кирпичей он положил, с момента последней проверки диспетчера. Диспетчер регулярно снимает показания строителей и в соответствии с ними регулирует границы участка стены каждого каменщика. Следовательно, необходимые потоки для реализации этого алгоритма: 1 поток диспетчер, N-1 - каменщики, где N - количество узлов в вычислительной системе.

1.2 Вывод аналитической части

В данном разделе рассмотрен принцип алгоритма построения кирпичной стены с диспетчером. Выделены необходимые для реализации алгоритма потоки: диспетчер и строители.

2. Конструкторская часть

В данном разделе представлены схемы алгоритмов. Так же будут описаны пользовательские структуры данных, приведены классы эквивалентности для тестирования реализуемого ПО.

2.1 Схема алгоритмов

На рисунке 2.1 показана схема алгоритмов работы диспетчера и каменщиков.

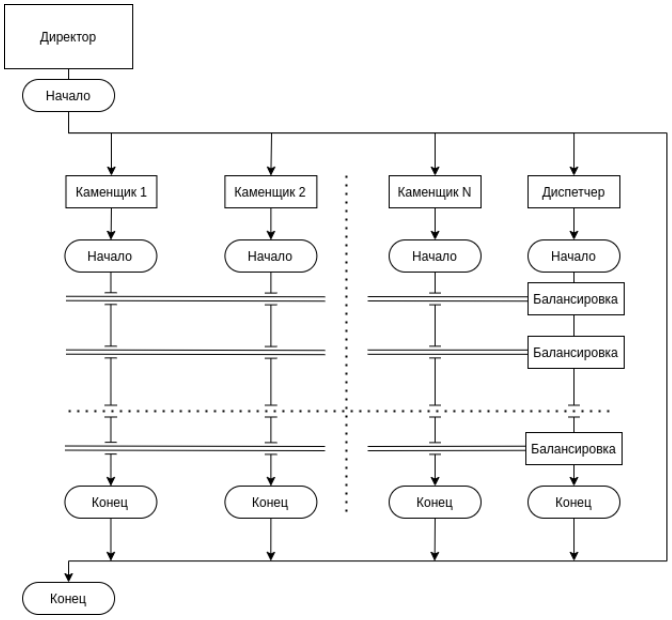

Рис. 2.1: Схема алгоритма полного перебора

2.2 Структуры данных

При реализации приведенных алгоритмов потребуются типы данных: матрица кирпичей, кирпич, параметры системы.

Кирпич:

1. название строителя

2. значение bool - установлен, или нет

Параметры:

1. ширина стены;
2. высота стены;
3. стартовая очередь кирпичей;
4. количество каменщиков;
5. границы участков каждого каменщика;
6. массив значений, где i - ое значение - количество установленных каменщиком.

2.3 Вывод конструкторской части

На основе данных, полученных в аналитическом разделе, была построена схема используемого алгоритма, выделены необходимые для реализации структуры данных.

3. Технологическая часть

3.1 Требования к ПО

Требования к программному обеспечению:

1. на вход подается ключ;

2. результат: значение, соответствующее ключу.

3.2 Выбор языка программирования

Был выбран язык go, поскольку он удовлетворяет требованиям лабораторной работы. Средой разработки выбрана Visual Studio Code. [3], [4], [5].

3.3 Структуры данных

На листинге 3.1, 3.2 представлено описание структуры кирпича и параметров системы.

Листинг 3.1: Структура кирпича

```
type brick_t struct{
  fixed bool
  builder_name int
}
```

Листинг 3.2: Параметры системы

```
type system_parameters_t struct{
  width int
  height int
  builders_count int
  len_limits len_limits_t

  flag_end_of_works bool

  bricks_matrix bricks_matrix_t
  builders_borders [][]int
  builders_works_status []int

  request_queue chan string

  dispetcher_time int
  builder_time int

  builders_wait_status []bool
}
```

3.4 Реализация алгоритмов

На листинге 3.5 представлена реализация алгоритма каменщиков.

Листинг 3.3: Реализация алгоритма каменщика ч. 1

```
func builder(system_parameters *system_parameters_t, builder_name int){
  print("BUILDER START\n")
  var x_res, y_res int
  var req string
  var con bool
  var time_to_sleep time.Duration

  ec := !system_parameters.flag_end_of_works
  for ;ec;{
    req = get_request(*system_parameters)
```

Листинг 3.4: Реализация алгоритма каменщика ч. 2

```
    req = get_request(*system_parameters)
    con = false
    for ;!con;{
      BIG_MUTEX.Lock()
      con = get_request_place(system_parameters, builder_name, &x_res, &
          y_res)
      if (con){
        system_parameters.bricks_matrix[x_res][y_res].fixed = true
        system_parameters.bricks_matrix[x_res][y_res].builder_name =
            builder_name
        system_parameters.builders_works_status[builder_name] +=1
        system_parameters.builders_wait_status[builder_name] = false
      } else {
        system_parameters.builders_wait_status[builder_name] = true
      }
      BIG_MUTEX.Unlock()

      if (con){
        time_to_sleep = time.Duration(len(req)*SLEEP_COEF*
            system_parameters.builder_time)/1000
        if (time_to_sleep > 1000*1000*1000*2) {
          time_to_sleep = 1000*1000*1000*2
        }
        time.Sleep(time_to_sleep)
        if (builder_name == 1){
          time.Sleep(time_to_sleep)
        }
      }
      BIG_MUTEX.Lock()
      draw_brick()
      BIG_MUTEX.Unlock()
      end_cond := system_parameters.flag_end_of_works
      if end_cond{
        break
      }
```

Листинг 3.5: Реализация алгоритма каменщика ч. 3

```
      }
    }
    ec = !system_parameters.flag_end_of_works
```

```
    }
}
```

На листинге 3.6 представлена реализация алгоритма диспетчера.

Листинг 3.6: Реализация алгоритма диспетчера

```
func dispetcher(system_parameters *system_parameters_t){
    print("DISPETCHER START\n")

    var con bool = true
    var new_borders [][] int
    for ;con;{

        var time_to_sleep = time.Duration(SLEEP_COEF * system_parameters.
            dispetcher_time)///1000

        time.Sleep(time_to_sleep)

        BIG_MUTEX.Lock()
        new_borders = generate_new_borders(system_parameters.
            builders_works_status,system_parameters.builders_wait_status,
          system_parameters.width)
        system_parameters.builders_borders = new_borders
        for i:=0;i<len(system_parameters.builders_works_status);i++{
          system_parameters.builders_works_status[i] = 0
        }
        BIG_MUTEX.Unlock()

        con = check_end_bricks_matrix(system_parameters.bricks_matrix)
        con = con && !system_parameters.flag_end_of_works
    }
    system_parameters.flag_end_of_works = true
}
```

Поскольку необходимо визуализировать стену Фокса, для наглядности, один из потоков будет работать в 2 раза медленее - 2 поток)

3.5 Тестирование

В таблице представлены тестовые данные 3.1.

Таблица 3.1 – Тесты

№	Ввод	Вывод
1	Стена 1*1	4
2	Стена 10*10	4
2	Стена 10*10	8
3	Стена 10*5	1
4	Стена 100*100	4
5	Стена 100*100	10

Тесты пройдены.

3.6 Вывод технологической части

Были реализованы исследуемые алгоритмы, программа прошла тесты и удовлетворяет требованиям.

4. Экспериментальная часть

Оценка качества работы алгоритмов.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

1. процессор: Intel® Core™ i3-7100U CPU @ 2.40GHz × 4;
2. память: 11,6 GiB;
3. операционная система: Ubuntu 20.04.1 LTS.

4.2 Примеры работы

На рисунке 4.1, 4.2, 4.3, 4.4 и 4.5 показан пример работы.

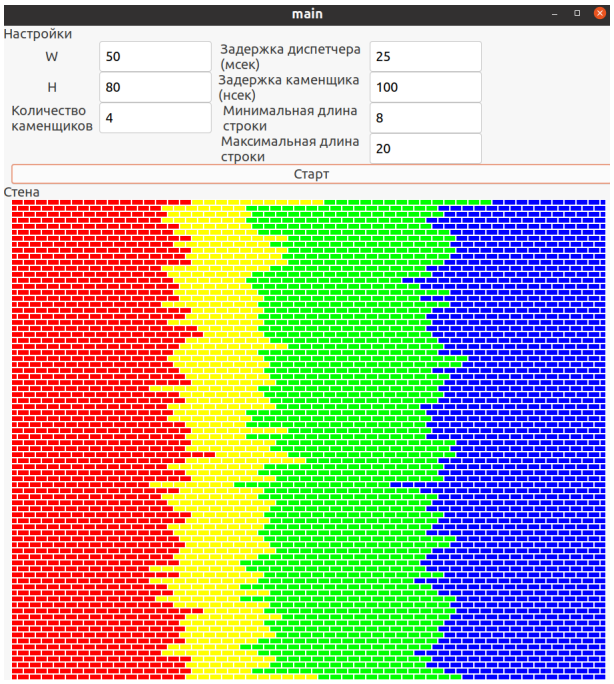

Рис. 4.1: Пример работы 1

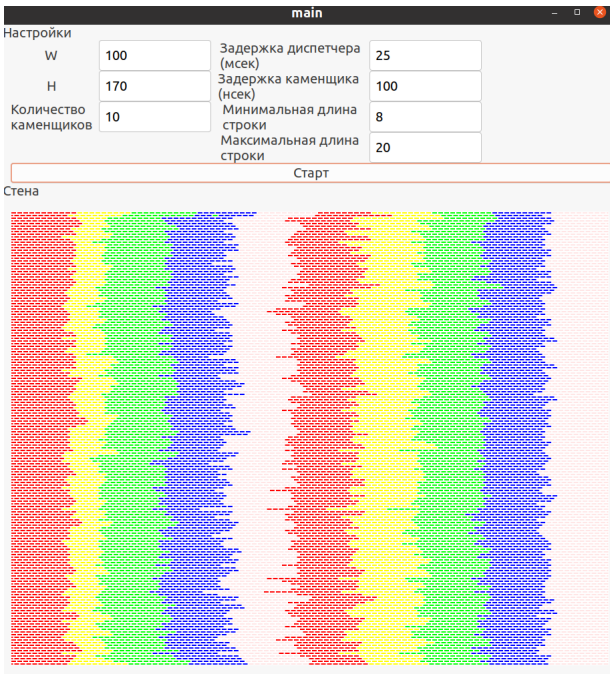

Рис. 4.2: Пример работы 2

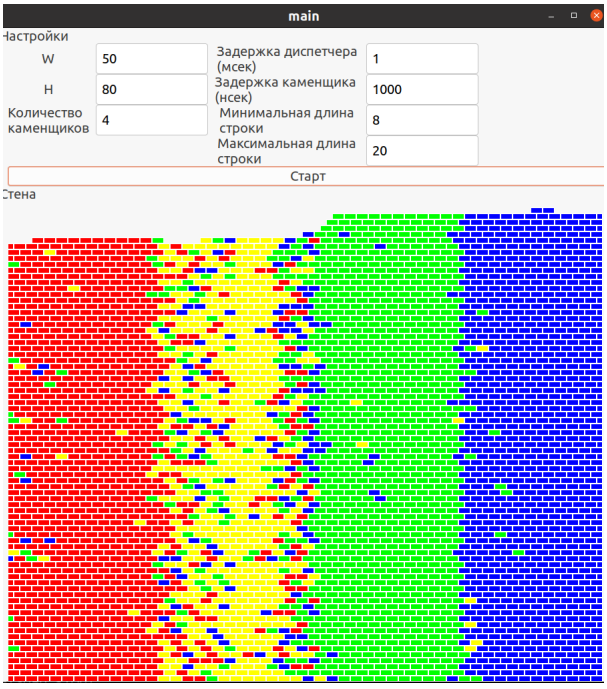

Рис. 4.3: Пример работы 3

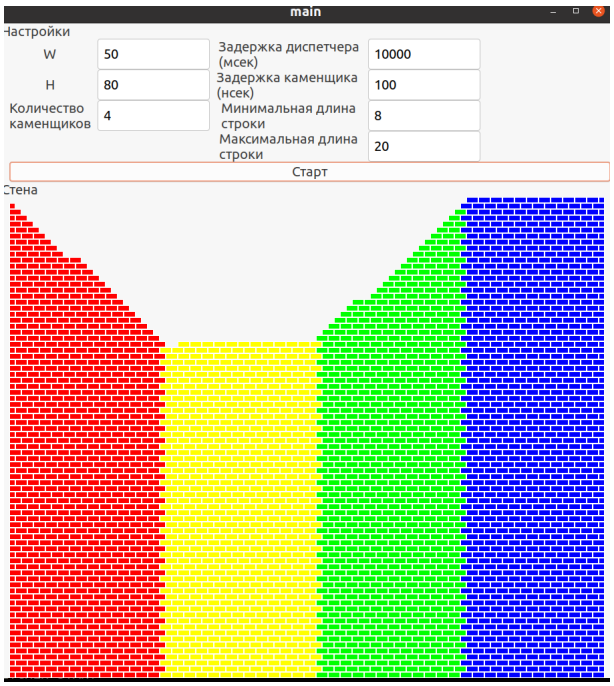

Рис. 4.4: Пример работы 4

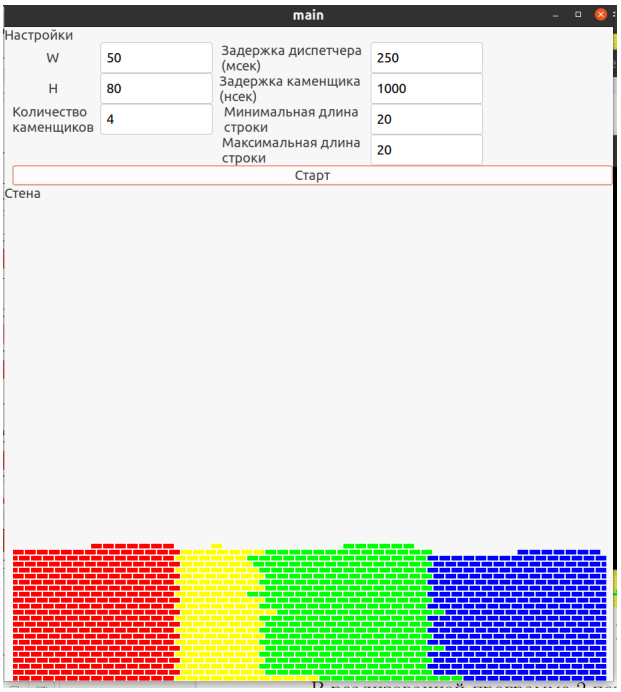

Рис. 4.5: Пример работы 5

В реализованной программе 2 поток работает в 2 раза медленее остальных. В примере 3 показана ситуация, когда диспетчер приходет слишком часто. В примере 4 показана ситуация, когда диспетчер приходет слишком редко. В обоих случаях, видно что 2 поток тормозит программу. В примере 5 видно, что потоки работают равномерно. В этом примере диспетчер работает около 1 раза на линию кирпичей.

4.3 Вывод экспериментальной части

В данном разделе было произведен анализ реализованного алгоритма. По результатам исследования было доказано, что алгоритм эффективно работает, если диспетчер обновляет границы участков каменщиков один раз за укладку линии.

Заключение

В данной работе были изучен алгоритмы построения стены Фокса. Получены практические навыки реализации алгоритма с динамической балансировкой нагрузки. Экспериментально подтверждены различия в эффективности алгоритмов с указанием лучших и худших случаев. Цель работы достигнута, решены поставленные задачи.

Список литературы

[1] В. Воеводин В. В. Воеводин Вл. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. – 608 с.

[2] Балансировка нагрузки в распределенных системах. Режим доступа:http://www.intuit.ru/department/algorithms/distrsa/9/.

[3] Go Documentation [Электронный ресурс]. Режим доступа: https://go.dev/doc/. Дата обращения: 29.01.2022.

[4] Linux – Getting Started [Электронный ресурс]. Режим доступа: https://linux.org. Дата обращения: 29.01.2022.

[5] Debian – универсальная операционная система [Электронный ресурс]. Режим доступа: https://www.debian.org/. Дата обращения: 29.01.2022.

[6] Макконнел. Дж. Анализ алгоритмов. Активный обучающий подход. – М. Режим доступа:https://www.linux.org.ru/. 2017. 267 с.

[7] Gratzer George A. More Math Into LaTeX. 4th изд. Boston: Birkhauser, 2007.