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

Тема Конвейерные вычисления

Студент Иммореева М.А.

Группа ИУ7-52Б

Оценка (баллы)

Преподаватель Строганов Д.В.

Москва — 2023 г.

Оглавление

Введение

Многопоточность – способность центрального процессора (CPU) или одного ядра в многоядерном процессоре одновременно выполнять несколько процессов или потоков, соответствующим образом поддерживаемых операционной системой. Смысл многопоточности — квазимногозадачность на уровне одного исполняемого процесса.

Параллельные вычисления часто используются для увеличения скорости выполнения программ. Однако приемы, применяемые для однопоточных машин, для параллельных могут не подходить. Конвейерная обработка данных является популярным приемом при работе с параллельными машинами.

При обработке данных также могут возникать ситуации, когда один набор данных необходимо обработать последовательно несколькими алгоритмами. В таком случае эффективно использовать конвейерную обработку данных, что позволяет на каждой следующей линии конвейера использовать данные, полученные с предыдущего этапа.

Целью данной лабораторной работы является реализация метода ковейерных вычислений.

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

1. Изучение основ конвейерных вычислений и конвейерной обработки данных.

2. Исследование подходов к реализации конвейерных вычислений на основе алгоритма поиска подстроки в строке Бойера–Мура.

3. Построение схем разработанного алгоритма.

4. Определение средств программной реализации.

5. Проведение тестирования реализованного алгоритма.

6. Реализация конвейерных вычислений с количеством линий не меньше трех.

7. Проведение сравнительного анализа времени выполнения параллельной и последовательной реализаций конвейерных вычислений.

8. Получение временной статистики программы, а именно время выполнения каждой ленты и время ожидания в очереди на выполнение.

1 Аналитическая часть

1.1 Цель и задачи

В данной лабораторной работе предлагается реализовать конвейерную обработку данных.

В качестве алгоритма был взят поиск подстроки в строке Бойера–Мура. Для каждого запроса будет сгенерирована строка со 100 символами. Ленты будут выполнять следующие задачи:

1. поиск подстроки в строке алгоритмом Бойера–Мура и запись числа найденных вхождений;
2. поиск подстроки в строке алгоритмом Бойера–Мура, запись в массив индексов вхождений и расчет количества сравнений букв во время работы алгоритма;
3. запись рассчитанных ранее данных в текстовый документ.

1.2 Конвейерная обработка данных

Конвейер – это устройство для непрерывного перемещения обрабатываемого изделия от одного рабочего к другому.

Конвейерное производство – система поточной организации производства на основе конвейера, при которой оно разделено на простейшие короткие операции, а перемещение деталей осуществляется автоматически.

В терминах программирования ленты конвейера представлены функциями, выполняющими над неким набором данных операции и передающие их на следующую ленту конвейера. Моделирование конвейерной обработки сочетается с технологией многопоточного программирования – под каждую ленту конвейера выделяется отдельный поток, все потоки работают в асинхронном режиме.

В данной лабораторной работе требуется выделить три задачи, которые будут последовательно обрабатываться на конвейерной ленте. Каждая задача будет последовательно проходить три этапа обработки. Благодаря распараллеливанию можно добиться того, что бы на всех трех этапах происходила обработка элемента.

Вывод

В данном разделе была рассмотрена предметная область работы и основополагающие материалы, которые в дальнейшем потребуются при реализации алгоритмов конвейерной обработки данных.

2 Конструкторская часть

В данном разделе приводится схема рассматриваемого алгоритма, определяются требования к программе.

2.1 Требования к программе

К вводу программы прилагаются данные требования: На вход подаются строка, вхождения которой ищутся в тексте файла, количество запросов.

На выходе программа выводит информацию о пройденных этапах каждого запроса.

2.2 Разработка алгоритмов

На рисунке 2.1 представлена схема работы конвейерной обработки. На рисунке 2.2 представлена схема работы главного потока, в котором просходит создание и запуск дочерних потоков на конвейерную обработку.

Рисунок 2.1 – Схема конвейерной обработки

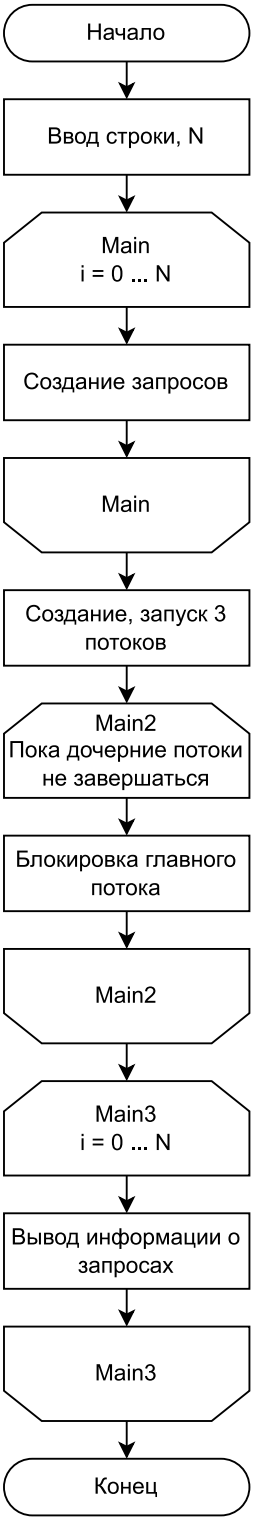


Рисунок 2.2 – Схема конвейерной обработки, главный поток

Вывод

В данном разделе были построены схемы работы алгоритма конвейерной обработки и главного потока, были приведены требования к программе.

3 Технологическая часть

В данном разделе будут приведены средства реализации и листинги кода.

3.1 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык программирования C++. Выбор обусловлен наличием библиотек для измерения времени, наличием инструментов для работы с потоками.

3.2 Сведения о модулях программы

Программа состоит из следующих модулей: main.cpp - главный файл программы, conveyor.cpp - файл с функциями для работы с конвейером, moor.cpp - файл с алгоритмом Бойера–Мура.

3.3 Реализация алгоритмов

В листингах 3.1, 3.2, 3.3, 3.4, приведены реализации алгоритмов.

Листинг 3.1 – Схема линейного конвейера.

```
void parse_linear(std::string str, size_t size, bool is_print)
{
    time_now = 0;
    to_print.clear();
    std::queue<request> q1;
    std::queue<request> q2;
    std::queue<request> q3;

    queues_t queues = {.q1 = q1, .q2 = q2, .q3 = q3};

    for (size_t i = 0; i < size; i++)
    {
        request res = generate(str);
        queues.q1.push(res);
    }

    for (size_t i = 0; i < size; i++)
    {
        request req = queues.q1.front();
        stage1_linear(req, i + 1, is_print);
        queues.q1.pop();
        queues.q2.push(req);

        req = queues.q2.front();
        stage2_linear(req, i + 1, is_print);
        queues.q2.pop();
        queues.q3.push(req);

        req = queues.q3.front();
        stage3_linear(req, i + 1, is_print);
        queues.q3.pop();

        if (is_print)
        {
            for (auto& log1: req.logger)
            {
                to_print.push_back(log1);
            }
        }
    }
```

```
41      if (is_print)
42      {
43          sort(to_print.begin(), to_print.end(), pred());
44          for (auto& one: to_print)
45          cout << one.data;
46      }
47  }
```

Листинг 3.2 – Главный поток параллельного конвейера

```
1  void parse_parallel(std::string str, size_t size, bool is_print)
2  {
3      to_print.clear();
4      t1.resize(size + 1);
5      t2.resize(size + 1);
6      t3.resize(size + 1);
7
8      for (size_t i = 0; i < size + 1; i++)
9      {
10         t1[i] = 0;
11         t2[i] = 0;
12         t3[i] = 0;
13     }
14
15     std::queue<request> q1;
16     std::queue<request> q2;
17     std::queue<request> q3;
18
19     queues_t queues = {.q1 = q1, .q2 = q2, .q3 = q3};
20
21
22     for (size_t i = 0; i < size; i++)
23     {
24         request res = generate(str);
25
26         q1.push(res);
27     }
28
29     std::thread threads[THREADS];
30
31     threads[0] = std::thread(stage1_parallel, std::ref(q1),
           std::ref(q2), std::ref(q3), is_print);
32     threads[1] = std::thread(stage2_parallel, std::ref(q1),
           std::ref(q2), std::ref(q3), is_print);
33     threads[2] = std::thread(stage3_parallel, std::ref(q1),
           std::ref(q2), std::ref(q3), is_print);
34
35     for (int i = 0; i < THREADS; i++)
36     {
37         threads[i].join();
```

```
38      }
39      if (is_print)
40      {
41          sort(to_print.begin(), to_print.end(), pred());
42          for (auto& one: to_print)
43          cout << one.data;
44      }
45  }
```

Листинг 3.3 – Лента параллельного конвейера

```
1  void stage1_parallel(std::queue<request> &q1, std::queue<request>
       &q2, std::queue<request> &q3, bool is_print)
2  {
3      int task_num = 1;
4
5      while(!q1.empty())
6      {
7          m.lock();
8          request req = q1.front();
9          m.unlock();
10
11         log_conveyor(req, task_num++, 1, is_print);
12
13         m.lock();
14         q2.push(req);
15         q1.pop();
16         m.unlock();
17     }
18 }
```

Листинг 3.4 – Лента последовательного конвейера

```
1  void log_linear(request &req, int task_num, int stage_num, bool
       is_print)
2  {
3      std::chrono::time_point<std::chrono::system_clock>
           time_start, time_end;
4      double start_res_time = time_now, res_time = 0;
5
6      time_start = std::chrono::system_clock::now();
7      if (stage_num == 1)
8      {
9          search(req.text, req.sub_str, req.ans);
10     }
11     else if (stage_num == 2)
12     {
13         search_detect(req.text, req.sub_str, req.inputs,
               req.comb);
14     }
15     else if (stage_num == 3)
16     {
17         log_in_file(req, task_num);
18     }
19
20     time_end = std::chrono::system_clock::now();
21
22     res_time =
           (std::chrono::duration_cast<std::chrono::nanoseconds>
       (time_end - time_start).count()) / 1e9;
24
25     time_now = start_res_time + res_time;
26
27     if (is_print)
28     {
29         log_s tmp;
30         tmp.data = "Task: "+ to_string(task_num) +", Tape: " +
               to_string(stage_num) + ", Start: "+ \
31         to_string(start_res_time) + ", End: " +
               to_string(start_res_time + res_time) + "\n";
32         tmp.start = start_res_time;
33         tmp.end = start_res_time + res_time;
34         req.logger.push_back(tmp);
```

```
35      }
36  }
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритма Бойера–Мура.

Таблица 3.1 – Тестирование функций

Строка	Паттерн	Ожидаемый результат
$abcdefabcdef$	abc	$0, 6$
abc	abc	0
$bcdeabcfabcdef$	abc	6
$bcdeacfabcdef$	abc	Пустой вывод
$bcdeacfabcdef$	Пустой ввод	ERR: pattern cant be empty
$hellowearetest3ing\_hellahell$	hel	$0, 29$

Алгоритм прошел проверку.

Вывод

Были разработан и протестирован конвейерный алгоритм.

4 Исследовательская часть

В данном разделе будет приведена постановка замера времени и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

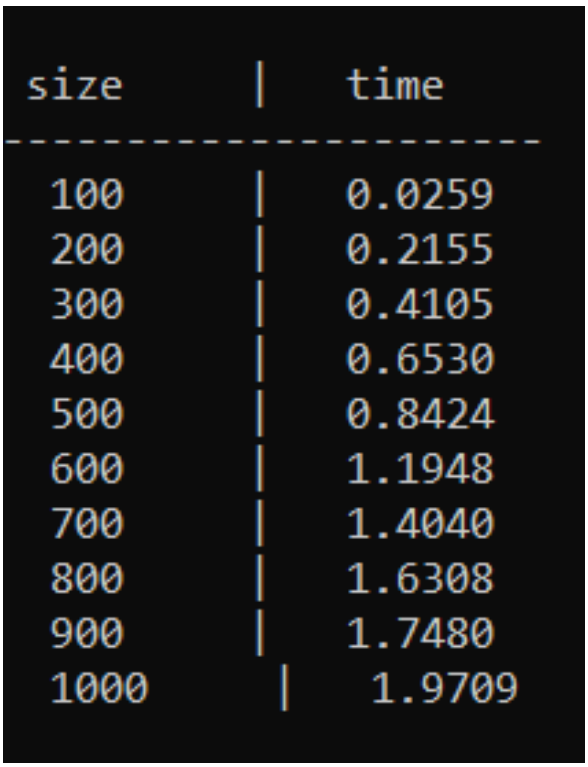
Технические характеристики устройства, на котором выполнялся замер времени. 64-разрядная операционная система, процессор x64. Память 16 Гб. Процессор Intel(R) Core(TM) i7–4700HQ CPU @ 2.40 ГГц.

Во время замера времени ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.2 Время выполнения алгоритмов

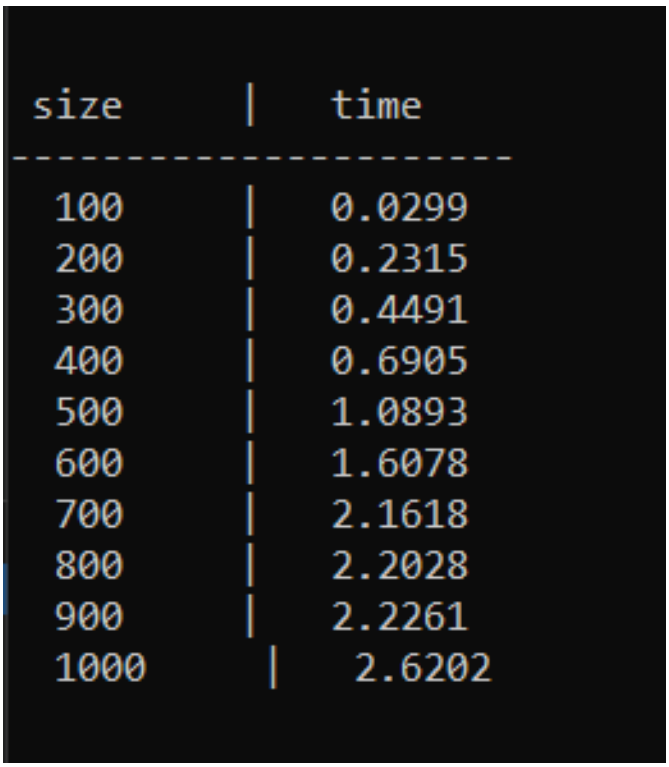
Алгоритмы замерялись при помощи функции *std* :: *chrono* :: *system_clock* из библиотеки *chrono* языка $C++$. Данная функция возвращает значение в долях секунды счетчика производительности, то есть часов с наибольшим доступным разрешением для измерения короткой длительности.

Замеры времени для каждого выполнения проводились 10 раз. В качестве результата взято среднее время работы алгоритма.



size	time
100	0.0259
200	0.2155
300	0.4105
400	0.6530
500	0.8424
600	1.1948
700	1.4040
800	1.6308
900	1.7480
1000	1.9709

Рисунок 4.1 – Замеры времени для параллельного конвейера



size	time
100	0.0299
200	0.2315
300	0.4491
400	0.6905
500	1.0893
600	1.6078
700	2.1618
800	2.2028
900	2.2261
1000	2.6202

Рисунок 4.2 – Замеры времени для последовательного конвейера

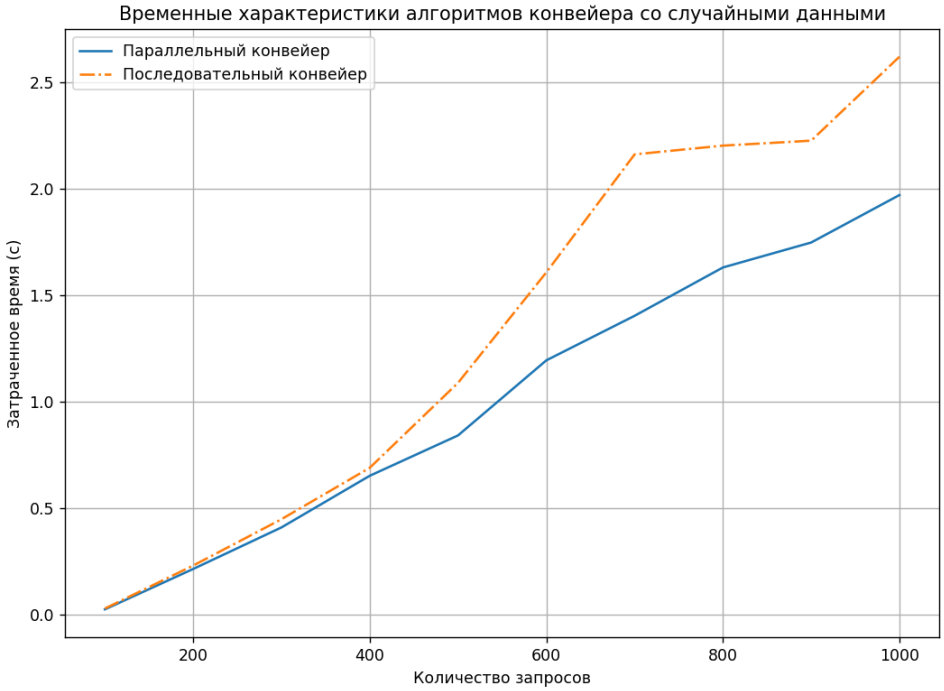


Рисунок 4.3 – Замеры времени для последовательного и параллельного конвейеров

Как и ожидалось, параллельная реализация конвейерной обработки выигрывает по времени выполнения у линейной реализации. Как видно из таблицы, линейная реализация начинает работать в разы медленнее при 500 задачах для обработки.

Вывод

В данном разделе было проведен сравнительный анализ количества затраченного процессорного времени линейной и параллельной реализаций конвейерной обработки, а результате которого было выяснено, что параллельная реализация значительно выигрывают по скорости у линейной при количестве строк равном 500.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

1. изучены основы конвейерной обработки данных на примере алгоритма поиска подстроки в строке Бойера–Мура;
2. выбран и описан метод обработки данных, которые будут сопоставлены методам конвейера;
3. описана архитектура ПО и реализованы линейная и параллельная реализации алгоритма конвейерной обработки;
4. произведен сравнительный анализ временных характеристик реализованных алгоритмов на экспериментальных данных.

Из исследовательской части сделан вывод – линейная реализация начинает работать в разы медленнее при 500 задачах для обработки.

Можно сделать вывод, что конвейерная обработка данных – это полезный инструмент, который уменьшает время выполнения программы за счет параллельной обработки данных. Самым эффективным временем считается время, когда все линии конвейера работают параллельно, обрабатывая свои задачи. Этот метод дает выигрыш по времени в том случае, когда выполняемые задачи намного больше по времени, чем время, затрачиваемое на реализацию конвейера (работу с потоками, перекладывание из очереди в очередь и тд).

Список использованных источников

[1] Волосова А. В. Параллельные методы и алгоритмы. – Москва: МАДИ, 2020.

[2] std::chrono::systemclock::now // cppreference.com URL: *https : //en.cppreference.com/w/cpp/chrono/system_clock/now* (дата обращения: 24.12.2023).