Отчет по лабораторной работе №5
по курсу "Анализ алгоритмов"
по теме "Распараллеливание вычислений"

Студент: Доктор А.А. ИУ7-53
Преподаватель: Волкова Л.Л., Строганов Ю.В.

2018 г.

Содержание

Введение

Процесс с двумя потоками выполнения на одном процессоре Поток выполнения (тред; от англ. thread — нить) — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Реализация потоков выполнения и процессов в разных операционных системах отличается друг от друга, но в большинстве случаев поток выполнения находится внутри процесса. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой момент времени). В качестве аналогии потоки выполнения процесса можно уподобить нескольким вместе работающим поварам. Все они готовят одно блюдо, читают одну и ту же кулинарную книгу с одним и тем же рецептом и следуют его указаниям, причём не обязательно все они читают на одной и той же странице [1].

На одном процессоре многопоточность обычно происходит путём временного мультиплексирования (как и в случае многозадачности): процессор переключается между разными потоками выполнения. Это переключение контекста обычно происходит достаточно часто, чтобы пользователь воспринимал выполнение потоков или задач как одновременное. В многопроцессорных и многоядерных системах потоки или задачи могут реально выполняться одновременно, при этом каждый процессор или ядро обрабатывает отдельный поток или задачу.

Потоки возникли в операционных системах как средство распараллеливания вычислений.

Параллельное выполнение нескольких работ в рамках одного интерактивного приложения повышает эффективность работы пользователя. Так, при работе с текстовым редактором желательно иметь возможность совмещать набор нового текста с такими продолжительными по времени операциями, как переформатирование значительной части текста, печать документа или его сохранение на локальном или удаленном диске. Еще одним примером необходимости распараллеливания является сетевой сервер баз данных. В этом случае параллелизм желателен как для обслуживания различных запросов к базе данных, так и для более быстрого выполнения отдельного запроса за счет одновременного просмотра различных записей базы. Именно для этих целей современные ОС предлагают механизм многопоточной обработки (multithreading). Понятию «поток» соответствует последовательный переход процессора от одной команды программы к другой. ОС распределяет процессорное время между потоками. Процессу ОС назначает адресное пространство и набор ресурсов, которые совместно используются всеми его потоками [2].

Создание потоков требует от ОС меньших накладных расходов, чем процессов. В отличие от процессов, которые принадлежат разным, вообще говоря, конкурирующим приложениям, все потоки одного процесса всегда принадлежат одному приложению, поэтому ОС изолирует потоки в гораздо меньшей степени, нежели процессы в традиционной мультипрограммной

системе. Все потоки одного процесса используют общие файлы, таймеры, устройства, одну и ту же область оперативной памяти, одно и то же адресное пространство. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к любому виртуальному адресу процесса, один поток может использовать стек другого потока. Между потоками одного процесса нет полной защиты, потому что, во-первых, это невозможно, а во-вторых, не нужно. Чтобы организовать взаимодействие и обмен данными, потокам вовсе не требуется обращаться к ОС, им достаточно использовать общую память — один поток записывает данные, а другой читает их. С другой стороны, потоки разных процессов по-прежнему хорошо защищены друг от друга.

Задачи работы

В ходе выполнения данной лабораторной работы, мной были выполнены следующие задачи:

1) научиться писать многопоточные программы;
2) применить полученные знания на практике, а именно переписать алгоритм Копперсмита — Винограда в несколько потоков;
3) провести замеры скорости работы однопоточной и многопоточной реализаций и проанализировать полученные результаты.

1 Аналитическая часть

В данном разделе приведено описание обычной реализации алгоритма Копперсмита — Винограда.

1.1 Алгоритм умножения матриц Винограда

Если посмотреть на результат умножения двух матриц, то видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее. Рассмотрим два вектора

V = (v1, v2, v3, v4) и

W = (w1, w2, w3, w4).

Их скалярное произведение равно:

V • W = v1w1 + v2w2 + v3w3 + v4w4.

Это равенство можно переписать в виде:

V • W = (v1 + w2)(v2 + w1) + (v3 + w4)(v4 + w3) - v1v2 - v3v4 - w1w2 - w3w4.

Кажется, что второе выражение задает больше работы, чем первое: вместо четырех умножений мы насчитываем их шесть, а вместо трех сложений - десять. Менее очевидно, что выражение в правой части последнего равенства допускает предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй. На практике это означает, что над предварительно обработанными элементами нам придется выполнять лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения.

Улучшения алгоритма

Возможны улучшения алгоритма с целью снизить его трудоемкость.

1. В 2010 Эндрю Стотерс усовершенствовал алгоритм до $O(n^{2.374})$.
2. В 2011 году Вирджиния Вильямс усовершенствовала алгоритм ещё раз — $O(n^{2.3728642})$.
3. В 2014 году Франсуа Ле Галль упростил метод Уильямса и получил новую улучшенную оценку $O(n^{2.3728639})$.

2 Конструкторская часть

В данном разделе размещены блоксхемы алгоритмов.

2.1 Разработка реализаций алгоритмов

Ниже приложены блоксхемы алгоритмов решения поставленных задач На рисунках 1 и 2 представлена реализация однопоточного алгоритма Винограда.

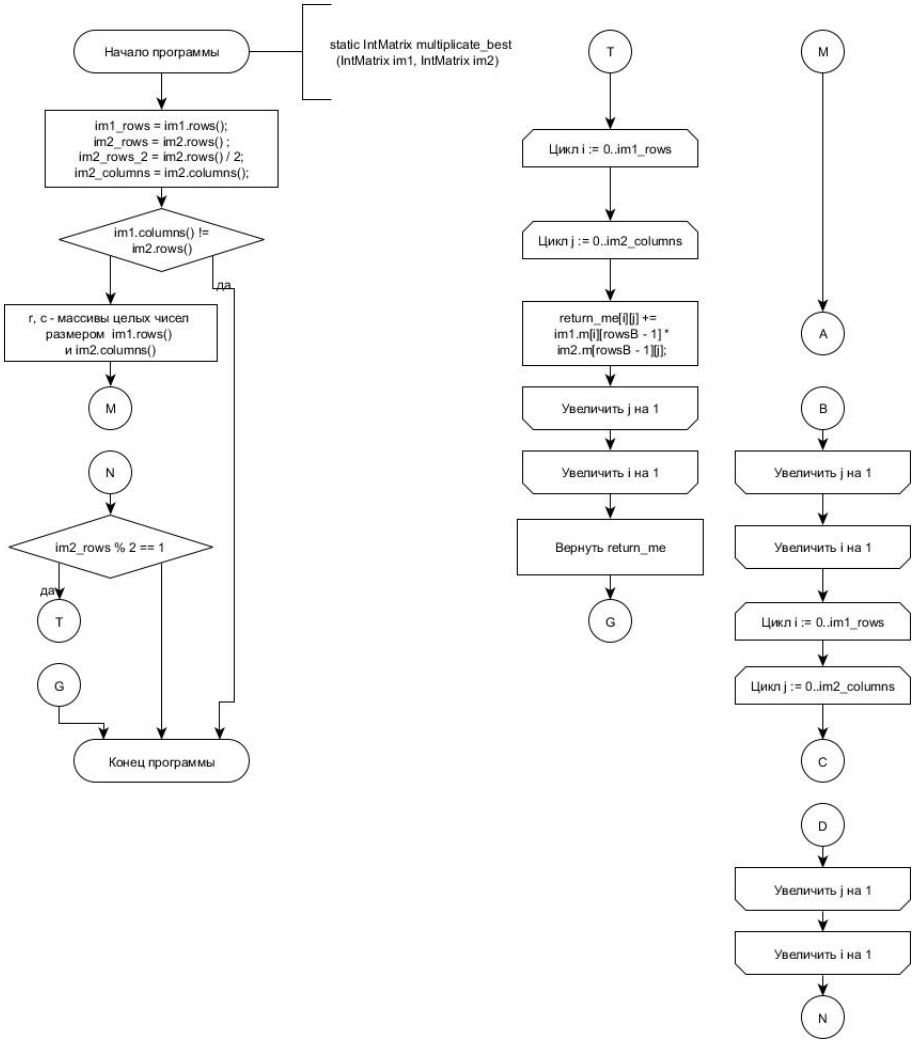

Рис. 1: Виноград с оптимизациями, часть 1

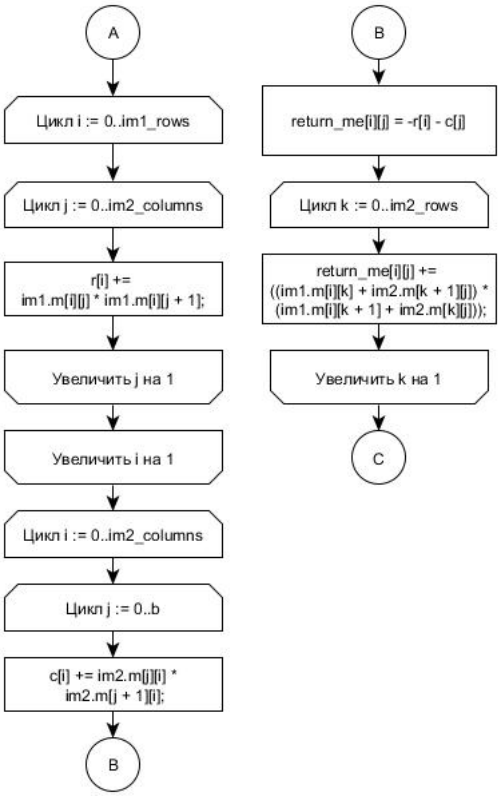

Рис. 2: Виноград с оптимизациями, часть 2

3 Технологическая часть

В данном разделе будут приведены:

1) средства реализации;

2) листинг кода;

3.1 Средства реализации

Для реализации программ мною был выбрал язык программирования - C++, поскольку имею опыт работы с ним. Среда разработки - Qt. Для работы с потоками использовались библиотеки thread и mutex. Однако поскольку потоки не использовали разделяемую память, то использование семафоров не понадобилось, если не считать их использование в выводе данных для отладки. Для реализаций матриц использовалась библиотека vector Отключение оптимизации производилось с помощью директивы препроцессора pragma optimize(, off). Время работы процессора замерялось с помощью функции, продемонстрированной в листинге 1. Эта функция в отличие от встроенной функции таймера, способна считать реальное про-

цессорное время работы программы в тиках [3]. Для ее работы была подключена библиотека time.h.

Листинг 1: Функция замера процессороного времени

```
1 unsigned long long tick(void)
2 {
3    unsigned long long d;
4    __asm__ __volatile__ ("rdtsc" : "=A"     (d));
5    return d;
6 }
7
```

3.2 Реализация алгоритмов

В листинге 2 представлена реализация однопоточной реализации алгоритма.

Листинг 2: Алгоритм Виноградова с оптимизациями

```
#pragma optimize("", off)
  static IntMatrix multiplicate_best(IntMatrix im1, IntMatrix im2) {

    if (im2.rows() != im1.columns())
       return (IntMatrix());
    IntMatrix return_me(im1.rows(), im2.columns());

    std::vector<int> r(im1.rows());
    std::vector<int> c(im2.columns());

    size_t im1_rows = im1.rows();
    size_t im2_rows_2 = im2.rows() / 2;
    size_t im2_columns = im2.columns();

    for (size_t i = 0; i < im1_rows; i++) {
       for (size_t j = 0; j < im2_rows_2; j++) {
          r[i] += im1.m[i][j] * im1.m[i][j + 1];
       }
    }

    for (size_t i = 0; i < im2_columns; i++) {
       for (size_t j = 0; j < im2_rows_2; j++) {
          c[i] += im2.m[j][i] * im2.m[j + 1][i];
       }
    }

    for (size_t i = 0; i < im1_rows; i++) {
       return_me.m[i].clear();
       for (size_t j = 0; j < im2_columns; j++) {
          return_me.m[i].push_back(0);
          return_me.m[i][j] = -r[i] - c[j];
          for (size_t k = 0; k < im2_rows_2; k++) {
             return_me.m[i][j] += ((im1.m[i][k] + im2.m[k + 1][j]) *
                                   (im1.m[i][k + 1] + im2.m[k][j]));
          }
       }
    }

    size_t im2_rows = im2.rows();
    if (im2_rows % 2) {
       for (size_t i = 0; i < im1_rows; i++) {
          for (size_t j = 0; j < im2_columns; j++) {
             im2_columns *
             return_me.m[i][j] += im1.m[i][im2_rows - 1] * im2.m[im2_rows - 1][
    j];
          }
       }
    }
```

```
48      return return_me;
49  }
50
```

Для реализации многопоточной реалиазация функция была разбита на 3 части. В листинге 3 показана первая часть, отвечающая за создание вспомогательной строки. Вторая функция 4 создает вспомогательный столбец. Непосредственно умножение матриц происходит в крайней функции 5.

Листинг 3: Создание вспомогательной строки

```
1   #pragma optimize("", off)
2    void work_1_3(IntMatrix im1,
3                  size_t im1_rows_begin,
4                  size_t im1_rows,
5                  size_t im2_rows_2,
6                  std::vector<int> &row){
7        size_t i_begin = im1_rows_begin;
8       size_t i_size = im1_rows;
9       size_t j_size = im2_rows_2;
10
11      for (size_t i = i_begin;u i < i_size; i++) {
12         for (size_t j = 0; j < j_size; j++) {
13            row[i] += im1.m[i][j] * im1.m[i][j + 1];
14         }
15      }
16    }
17
```

Листинг 4: Создание вспомогательного столбца

```
1 #pragma optimize("", off)
2  void work_2_3(IntMatrix im2,
3              size_t im2_columns_begin,
4              size_t im2_columns,
5              size_t im2_rows_2,
6              std::vector<int> &column){
7     size_t i_begin = im2_columns_begin;
8     size_t i_size = im2_columns;
9     size_t j_size = im2_rows_2;
10
11    for (size_t i = i_begin; i < i_size; i++) {
12       for (size_t j = 0; j < j_size; j++) {
13          column[i] += im2.m[j][i] * im2.m[j + 1][i];
14       }
15    }
16  }
17
```

Листинг 5: Перемножение матриц

```
1 #pragma optimize("", off)
2 void work_3_3(IntMatrix im1, IntMatrix im2,
3           std::vector<int> c,
```

```
4           std::vector<int> r,
5           size_t im1_rows_begin,
6           size_t im1_rows,
7           size_t im2_columns,
8           size_t im2_rows_2,
9           IntMatrix &return_me)
10{
11
12  size_t im2_rows = im2.rows();
13  size_t i_begin = im1_rows_begin;
14  size_t i_size = im1_rows;
15  size_t j_size = im2_columns;
16  size_t k_size = im2_rows_2;
17
18  for (size_t i = i_begin; i < i_size; i++) {
19     return_me.m[i].clear();
20
21     for (size_t j = 0; j < j_size; j++) {
22        return_me.m[i].push_back(0);
23
24        return_me.m[i][j] = -r[i] - c[j];
25
26        for (size_t k = 0; k < k_size; k++) {
27           return_me.m[i][j] += (
28                 (im1.m[i][k] + im2.m[k + 1][j]) *
29                 (im1.m[i][k + 1] + im2.m[k][j])
30              );
31        }
32        if (im2_rows % 2) {
33           return_me.m[i][j] += im1.m[i][im2_rows - 1] *
34                 im2.m[im2_rows - 1][j];
35        }
36     }
37  }
38}
39
```

В листинге 6 представлена реализация двупоточной версии алгоритма. 4-ех, 6-и, 8-и поточные реализации сделаны аналогичным образом.

Листинг 6: Двупоточная реализация алгоритма Винограда

```
#pragma optimize("", off)
IntMatrix multi2(IntMatrix im1, IntMatrix im2) {

if (im2.rows() != im1.columns())
   return (IntMatrix());

IntMatrix return_me(im1.rows(), im2.columns());

std::vector<int> r(im1.rows());
std::vector<int> c(im2.columns());

size_t im1_rows = im1.rows();
size_t im2_rows_2 = im2.rows() / 2;
size_t im2_columns = im2.columns();

thread thread1_1(work_1_3,
                 im1,
                 0, im1_rows,
                 im2_rows_2,
                 ref(r));

thread thread2_1(work_2_3, im2,
                 0, im2_columns,
                 im2_rows_2,
                 ref(c));
thread1_1.join();
thread2_1.join();

thread thread3_1(work_3_3, im1, im2
                 c, r, 0, im1_rows / 2, im2_columns, im2_rows_2,
                 ref(return_me));

thread thread3_2(work_3_3, im1, im2,
                 c, r, im1_rows / 2, im1_rows, im2_columns, im2_rows_2,
                 ref(return_me)));

thread3_1.join();
thread3_2.join();

return return_me;
}

```

4 Экспериментальная часть

В данном разделе будет сравнительный анализ реализаций алгоритма Винограда на основе экспериментальных данных. Во время проведения замеров использовался ноутбук с 2 физическими ядрами и 4 логическими процессорами.

4.1 Сравнительный анализ

Для экспериментов использовались массивы, размер которых варьируется от 100 до 500 с шагом 100. Количество повторов каждого эксперимента = 100. Результат одного эксперимента рассчитывается как средний из результатов проведенных испытаний с одинаковыми входными данными. Результаты замеров времени сведены в табл. 1. Таблица состоит из 6 столбцов. Первый столбец представляет собой количество элементов. Второй - обычный оптимизированный алгоритм ВИнограда. Третий, четвертый, пятый и шестой - 2, 4, 6 и 8 поточные реализации. Те же результаты представлены на рис. 3.

Таблица 1: Результаты сравнения версий

Количество элементов	1 поток (в тиках)	2 потока (в тиках)	4 потока (в тиках)	6 потоков (в тиках)	8 потоков (в тиках)
100	42120301	28547882	25954733	30014805	28076281
200	309628478	206121035	157828274	176815008	166488208
300	1018593267	640086297	518701316	556741606	571883017
400	2401139769	1483276424	1277150942	1295142955	1291652421
500	4670233016	2825425249	2442188430	2512812630	2486895374

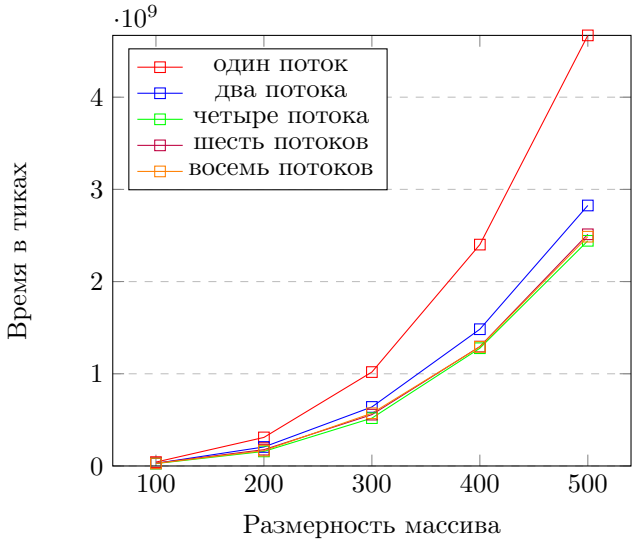

Рис. 3: График версий алгоритма

Вывод

Как и ожидалось, многопоточные программы работают быстрее однопоточных решений, причем разница достигает почти двух раз(зависит от нагруженности процессора. Чем больше нагрузка, тем меньше эффективность нескольких потоков, поскольку им приходится простаивать в очереди). Наиболее эффективным количеством потоков оказалось число 4, равное количеству логических процессоров. При дальнейшим увеличении количества потоков увеличения скорости нет, зато увеличивается затрачиваемое время, что связано с издержками ожидания процессорного времени для потоков.

Заключение

В ходе работы были изучена и реализована многопоточность. Поскольку было выбрано разделение алгоритма так, чтобы не было разделяемой памяти, использования семафоров не было. Был сделан вывод, что наиболее оптимальное количество потоков в программе должно соответствовать количеству логических процессоров используемого компьютера.

Список литературы

[1] kiranshobha01@gmail.com статья про устройство мультипоточности http://thekiransblog.blogspot.com/2010/02/multithreading.html

[2] Потоки и распараллеливание вычислений https://poznayka.org/s10704t1.html

[3] И.В. Ломовской. Курс лекций по языку программирования C, 2017