Оглавление

Введение

Целью данной лабораторной работы является описание параллельных конвейерных вычислений. Для достижения поставленной цели необходимо выполнить следующие задачи:

- описать организацию конвейерной обработки данных;
- реализовать программу, реализующую конвейер с количеством лент не менее трех в однопоточной и многопоточной среде;
- исследовать зависимость времени работы конвейера от количества потоков, на которых он работает.

1 Аналитическая часть

В этом разделе будут представлены описания алгоритмов умножения разреженных матриц, последовательного и параллельного.

1.1 Конвейерные вычисления

Способ организации процесса в качестве вычислительного конвейера (pipeline) позволяет построить процесс, содержащий несколько независимых этапов [1], на нескольких потоках. Выигрыш во времени достигается при выполнении нескольких задач за счет параллельной работы ступеней, вовлекая на каждом такте новую задачу или команду. Для контроля стадии используются три основные метрики, описанные ниже.

1. Время процесса - это время, необходимое для выполнения одной стадии.
2. Время выполнения - это время, которое требуется с момента, когда работа была выполнена на предыдущем этапе, до выполнения на текущем.
3. Время простоя - это время, когда никакой работы не происходит и линии простаивают.

Для того, чтобы время простоя было минимальным, стадии обработки должны быть одинаковы по времени в пределах погрешности. При возникновении ситуации, в которой время процесса одной из линий больше, чем время других в N раз, эту линию стоит распараллелить на N потоков.

1.2 Умножение разреженных матриц

Разреженная матрица [2] — в численном анализе и научных вычислениях, разреженная матрица или разреженный массив представляет собой матрицу, в которой большинство элементов равны нулю. Не существует строгого определения того, сколько элементов должно быть нулевым, чтобы матрица считалась разреженной, но общий критерий состоит

в том, что количество ненулевых элементов примерно равно количеству строк или столбцов. Напротив, если большинство элементов отличны от нуля, матрица считается плотной. Количество элементов с нулевым знаком, деленное на общее количество элементов, называют разреженностью матрицы.

При хранении разреженных матриц и манипулировании ими на компьютере необходимо использовать специализированные алгоритмы и структуры данных, которые используют разреженную структуру матрицы. Специализированные компьютеры были созданы для разреженных матриц, поскольку они распространены в области машинного обучения. Операции с использованием стандартных структур и алгоритмов с плотной матрицей медленны и неэффективны при применении к большим разреженным матрицам, поскольку обработка и память тратятся на нули. Разреженные данные по своей природе легче сжимать и, следовательно, требуют значительно меньше памяти. Некоторыми очень большими разреженными матрицами невозможно манипулировать с помощью стандартных алгоритмов плотных матриц.

Алгоритм умножения двух разреженных матриц является набором определенных шагов.

1. Создать матрицу результата.
2. Первый цикл проходит по строкам матриц.
3. Второй цикл идет до тех пор, пока в массиве значений левой матрицы не закончится рассматриваемая строка. В этом цикле выполняются следующие действия:

   (a) Пройти по всем элементам правой матрицы и умножить их на элементы, рассматриваемой на данном шаге строки. Прибавить результат умножения к элементу массива промежуточных значений, номер элемент а определяем по значению столбца правой матрицы, в котором находится, рассматриваемый элемент.

   (b) После умножения всех значений правой матрицы сохранить значение массива в вектор значений матрицы результата. А так же обновить вектор столбцов для каждого элемента.

(c) Добавить размер вектора столбцов в вектор строк матрицы результата.

4. После завершения первого цикла в матрице результата находится результат умножения переданных двух матриц. Вернуть из метода матрицу ответа.

Вывод

Алгоритм умножения разреженных матриц работает независимо от чтения и вывода из файла, что дает возможность реализовать конвейер.

2 Конструкторская часть

В этом разделе будут приведены схемы алгоритмов и описаны используемые типы данных.

2.1 Разработка алгоритмов

На рисунке 2.1 представлена схема алгоритма умножения разреженных матриц.

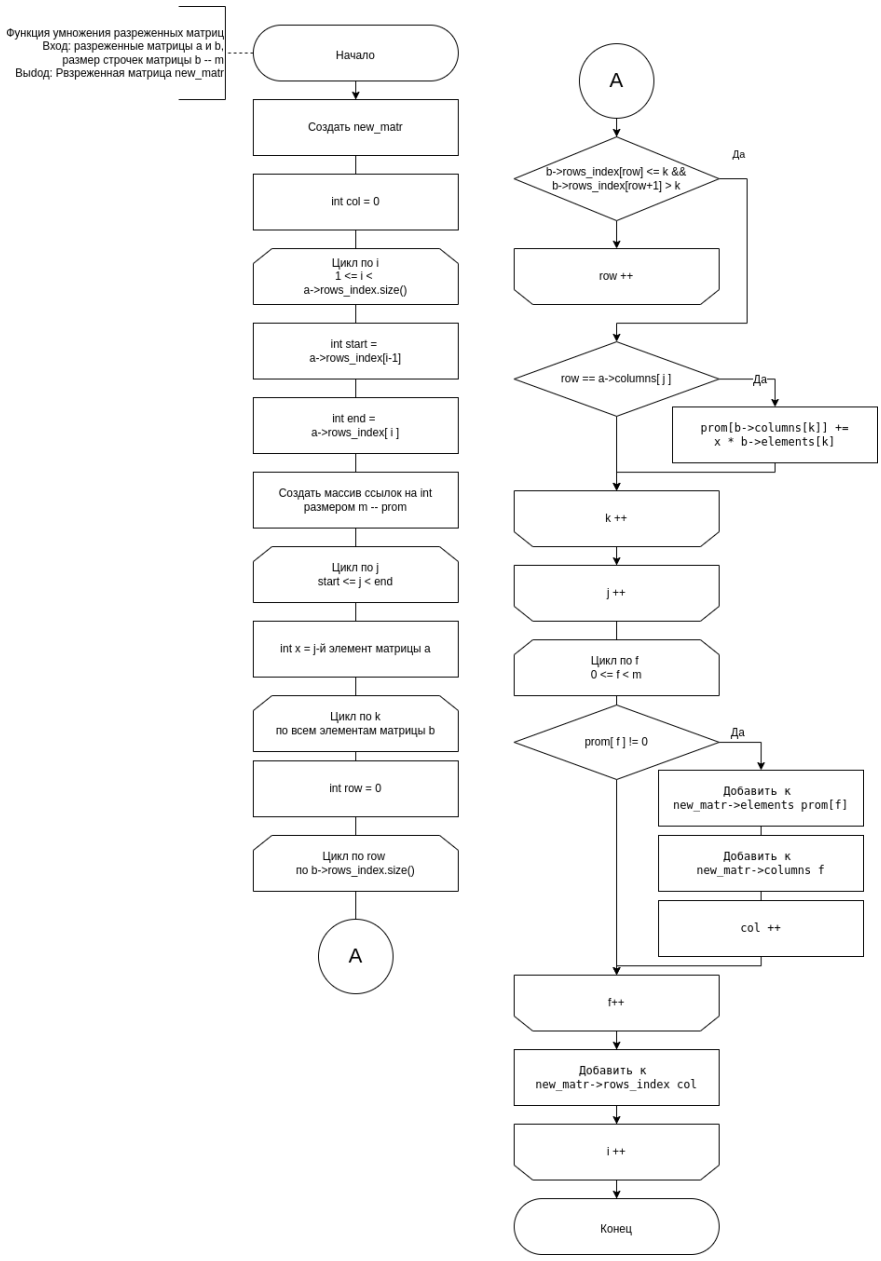

Рисунок 2.1 – Схема последовательного алгоритма умножения разреженных матриц

На рисунке 2.2 представлена схема конвейерной линии.

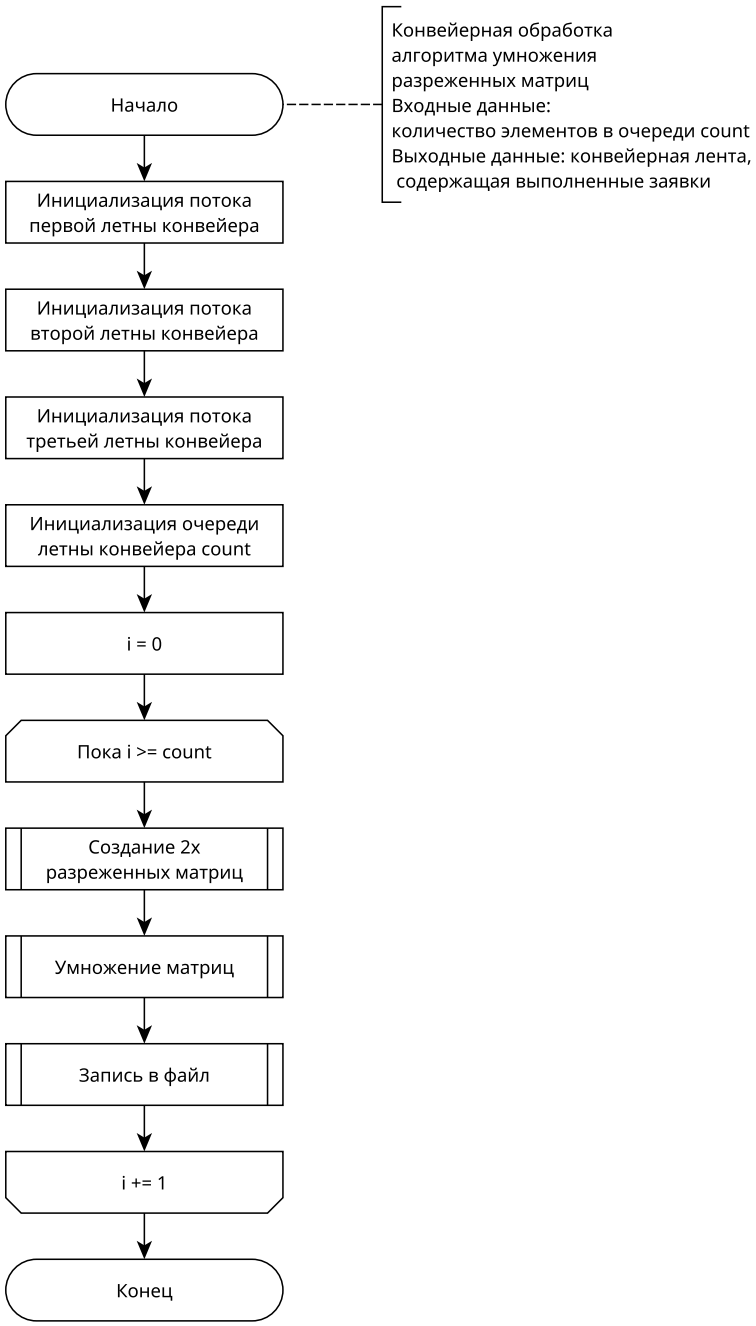

Рисунок 2.2 – Схема конвейерного алгоритма умножения разреженных матриц

Вывод

Были разработаны схемы последовательного и параллельного алгоритмов умножения разреженных матриц.

3 Технологическая часть

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

3.1 Требования к программному обеспечению

К программе предъявляется ряд требований:

— на вход подается количество умножений;

— используется параллелизм программы;

— возможно измерение реального времени;

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык программирования Си++ [3].

В данном языке есть все требующиеся для данной лабораторной инструменты разработки.

Процессорное время работы реализаций алгоритмов было замерено с помощью функции high_resolution_clock::now() из библиотеки <chrono> [4].

3.3 Реализация алгоритмов

В листинге 3.1 представлена реализация алгоритма умножения разреженных матриц.

Листинг 3.1 – Реализация последовательного алгоритма умножения разреженных матриц

```
1 matr *mutl(matr *a, matr *b, const int m)
```

```
2 {
3     matr *new_matr = new(matr); // 1
4     new_matr->rows_index.push_back(0);
5     int col = 0;
6     for (int i = 1; i < int(a->rows_index.size()); i++) // 2
7     {
8         int start = a->rows_index[i-1];
9         int end = a->rows_index[i];
10        int *prom = (int *)calloc(m, sizeof(int));
11        for (int j = start; j < end; j++) { // 3
12            int x = a->elements[j];
13            for (int k = 0; k < int(b->elements.size()); k++) {
14                int row;
15                for (row = 0; row < int(b->rows_index.size() -
                      1); ++row) {
16                    if (b->rows_index[row] <= k &&
                          b->rows_index[row + 1] > k)
17                        break;
18                }
19                if (row == a->columns[j])
20                prom[b->columns[k]] += x * b->elements[k]; // 3.a
21            }
22        }
23        for (int f = 0; f < m; f++) // 3.b
24        if (prom[f]) {
25            new_matr->elements.push_back(prom[f]);
26            new_matr->columns.push_back(f);
27            col++;
28        }
29        new_matr->rows_index.push_back(col); // 3.c
30        free(prom);
31    }
32    return new_matr; // 4
33 }
```

В листинге 3.2 представлена реализация синхронного конвейера.

Листинг 3.2 – Реализация функции параллелизации умножения разреженных матриц

```
1
2 pipeTask* gener(pipeTask* pt) {
3     pt->start_generate = chrono::high_resolution_clock::now();
```

```
4
5       auto k1 = generate();
6       pt->m1 = get_csrrepresent_m(k1);
7       k1 = generate();
8       pt->m2 = get_csrrepresent_m(k1);
9
10      pt->stop_generate = chrono::high_resolution_clock::now();
11
12      return pt;
13  }
14
15  pipeTask* multic(pipeTask* pt) {
16      pt->start_mul = chrono::high_resolution_clock::now();
17
18      pt->mrez = mutl((pt->m1), (pt->m2), N);
19
20      pt->stop_mul = chrono::high_resolution_clock::now();
21
22      return pt;
23  }
24
25  pipeTask* write(pipeTask* pt) {
26      pt->start_write = chrono::high_resolution_clock::now();
27
28      ofstream out("out.txt");
29
30
31      out << '\n';
32      for (int j = 0; j < int(pt->mrez->rows_index.size()); j++)
33      out << pt->mrez->rows_index[j] << ' ';
34      out << '\n';
35      for (int j = 0; j < int(pt->mrez->columns.size()); j++)
36      out << pt->mrez->columns[j] << ' ';
37      out << '\n';
38      for (int j = 0; j < int(pt->mrez->elements.size()); j++)
39      out << pt->mrez->elements[j] << ' ';
40      out << '\n';
41
42      out.close();
43
44      pt->stop_write = chrono::high_resolution_clock::now();
```

```
45
46      return pt;
47  }
48
49
50  void gener_conv(queue* qu, queue* qu2, int l) {
51      int ll = 0;
52      while (1) {
53          if (qu->queue.size() > 0) {
54              auto k = gener(qu->queue[0]);
55              qu->queue.erase(qu->queue.begin());
56              qu2->queue.push_back(k);
57              ll++;
58          }
59
60          if (ll == l)
61          break;
62      }
63  }
64
65  void multic_conv(queue* qu, queue* qu3, int l) {
66      int ll = 0;
67      while (1) {
68          if (qu->queue.size() > 0) {
69              auto k = multic(qu->queue[0]);
70              qu->queue.erase(qu->queue.begin());
71              qu3->queue.push_back(k);
72              ll++;
73          }
74
75          if (ll == l)
76          break;
77      }
78  }
79
80  void write_conv(queue* qu, queue* quf, int l) {
81      int ll = 0;
82      while (1) {
83          if (qu->queue.size() > 0) {
84              auto k = write(qu->queue[0]);
85              qu->queue.erase(qu->queue.begin());
```

```
86              quf->queue.push_back(k);
87              ll++;
88          }
89
90          if (ll == l)
91          break;
92      }
93  }
94
95  queue* pipeline(int count) {
96      vector<thread> threads(3);
97
98      queue* qu1 = new(queue);
99      queue* qu2 = new(queue);
100     queue* qu3 = new(queue);
101     queue* quf = new(queue);
102
103     qu1->stop = false;
104     qu2->stop = false;
105     qu3->stop = false;
106
107     threads[0] = thread(gener_conv, qu1, qu2, count);
108     threads[1] = thread(multic_conv, qu2, qu3, count);
109     threads[2] = thread(write_conv, qu3, quf, count);
110
111     for (int i = 0; i < count; i++) {
112         auto pt = new(pipeTask);
113         pt->n = i;
114         qu1->queue.push_back(pt);
115     }
116
117     for (auto& thr: threads)
118     thr.join();
119
120     return quf;
121 }
122 }
123 }
```

Таблица 3.1 – Тестовые случаи

N	Матрица 1	Матрица 2	Результат
1	$\begin{pmatrix} 1 & 1 & 1 \\ 5 & 5 & 5 \\ 2 & 2 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 3 \\ 15 \\ 6 \end{pmatrix}$
2	$\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$
3	$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$
4	$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 2 \end{pmatrix}$	$\begin{pmatrix} 4 \end{pmatrix}$
5	$\begin{pmatrix} 1 & -2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} -1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$	$\begin{pmatrix} 0 & 4 & 6 \\ 4 & 12 & 18 \\ 4 & 12 & 18 \end{pmatrix}$

3.4 Функциональные тесты

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы сортировки. Тесты пройдены успешно.

Вывод

Написано и протестировано программное обеспечение для поставленной задачи.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программ, постановка эксперимента и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: Manjaro xfce [5] Linux [6] x86_64;
- память — 8 Гб;
- мобильный процессор AMD Ryzen™ 7 3700U @ 2.3 ГГц [7].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы.

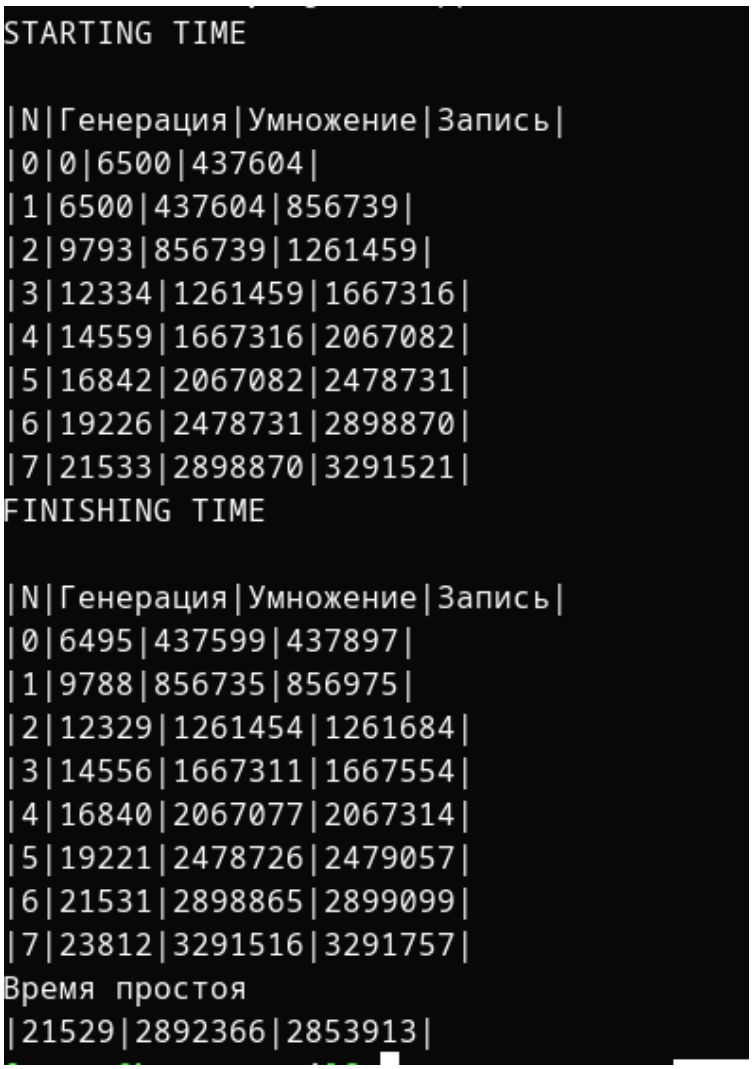

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения реализации алгоритмов

Время работы реализации алгоритмов измерялось при помощи функции chrono::high_resolution_clock::now() из библиотеки <chrono> языка C++.

№	Начало обработки заявки					
	Параллельно, мс.			Синхронно, мс.		
	1	2	3	1	2	3
1	0	3050	435427	0	4546	455827
2	3050	435427	828247	456041	457950	865352
3	8130	828247	1228188	865554	867585	1278679
4	12261	1228188	1638834	1278873	1281023	1680580
5	14427	1638834	2039991	1680808	1682876	2092687
6	16579	2039991	2441431	2092876	2094858	2497214
7	18724	2441431	2858998	2497453	2499488	2907603
8	20974	2858998	3242625	2907837	2909908	3313635

Таблица 4.1 – Замеры времени работы на очереди размером 8

Из таблицы можно сделать вывод, что рапараллеленый конвейер выполняет работу на 10% быстрее, чем последовательный.

Вывод

В данном разделе были сравнены алгоритмы по времени. Выявлено, что конвейерная обработка быстрее последовательной на 10%.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

— описана организация конвейерной обработки данных;

— реализованны программа, реализующая конвейер с тремя лентами;

— исследована зависимость времи работы конвейера от количества потоков, на которых он работает.

В данном были сравнены алгоритмы по времени. Выявлено, что конвейерная обработка быстрее последовательной на 10%.

Поставленная цель достигнута: описаны параллельные конвейерные вычисления.

Список использованных источников

1. Pipeline Processing in digital logic design [Электронный ресурс]. Режим доступа: https://www.fullchipdesign.com/pipeline_space_time_architecture.htm (дата обращения: 15.12.2022).

2. Писсарецки С. Технология разреженных матриц. 1984. Т. 3. с. 355.

3. Welcome to Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 04.09.2021).

4. time — Time access and conversions [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html#functions (дата обращения: 04.09.2021).

5. Manjaro [Электронный ресурс]. Режим доступа: https://manjaro.org/ (дата обращения: 03.10.2022).

6. Linux – Википедия [Электронный ресурс]. Режим доступа: https://ru.wikipedia.org/wiki/Linux (дата обращения: 04.10.2022).

7. Мобильный процессор AMD Ryzen™ 7 3700U [Электронный ресурс]. Режим доступа: https://www.amd.com/ru/products/apu/amd-ryzen-7-3700u (дата обращения: 04.10.2022).