Государственное образовательное учреждение высшего профессионального образования
«Московский государственный технический университет имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)

Лабораторная работа №5

по курсу «Анализ алгоритмов»

Конвейерная обработка

Выполнил: Сорокин А.П., гр. ИУ7-52Б

Преподаватели: Волкова Л.Л., Строганов Ю.В.

Москва, 2019 г.

Оглавление

Введение

Сам термин «конвейер» пришёл из промышленности, где используется аналогичный принцип работы — материал автоматически подтягивается по ленте конвейера к рабочему, который осуществляет с ним необходимые действия, следующий за ним рабочий выполняет свои функции над получившейся заготовкой, следующий делает еще что-то, таким образом, к концу конвейера цепочка рабочих полностью выполняет все поставленные задачи, не срывая, однако, темпов производства. Например, если на самую медлительную операцию затрачивается одна минута, то каждая деталь будет сходить с конвейера через одну минуту.

Идея заключается в разделении обработки компьютерной инструкции на последовательность независимых стадий с сохранением результатов в конце каждой стадии. Это позволяет управляющим цепям процессора получать инструкции со скоростью самой медленной стадии обработки, однако при этом намного быстрее, чем при выполнении эксклюзивной полной обработки каждой инструкции от начала до конца.

1. Аналитический раздел

В данном разделе будет описан принцип конвейерной обработки.

1.1 Цель и задачи

Цель лабораторной работы: изучений конвейрной обработки. Для достижения этой цели были поставлены следующие задачи:

1. разработка и реализация алгоритмов
2. исследование работы конвейерной обработки с использование многопоточности и без
3. описание и обоснование полученных результатов

1.2 Конвейерная обработка данных

Если задача заключается в применении одной последовательности операций ко многим независимым элементам данных, то можно организовать распараллеленный конвейер. Здесь можно провести аналогию с физическим конвейером: данные поступают с одного конца, подвергаются ряду операций и выходят с другого конца. Для того, чтобы распределить работу по принципу конвейерной обработки данных, следует создать отдельный поток для каждого участка конвейера, то есть для каждой операции. По завершении операции элемент данных помещается в очередь, откуда его забирает следующий поток. В результате поток, выполняющий первую операцию, сможет приступить к обработке следующего элемента, пока второй поток трудится над первым элементом. Конвейеры хороши также тогда, когда каждая операция занимает много времени; распределяя между потоками задачи, а не данные, мы изменяем качественные показатели производительности [1].

1.3 Вывод

В данном разделе был описан принцип конвейерной обработки.

2. Технологический раздел

В данном разделе будет предъявлены требования к разрабатываемому программному обеспечению, средства, использованные в процессе разработки для реализации поставленных задач, а также представлен листинг кода рпограммы.

2.1 Требования к программному обеспечению

Программное обеспечение должно реализовывать линейную, конвейерную обработку данных. Пользователь должен иметь возможность вводить количество объектов, которые будут обрабатываться.

2.2 Средства реализации

Для реализации программы был использован язык C++ [2]. Для замера процессорного времени была использована функция rdtsc() из библиотеки stdrin.h.

2.3 Листинг кода

В листинге 2.1 представлена реализация линейной и конвейерной обработки матриц.

Листинг 2.1: Реализация линейной и конвейерной обработки матрицы

```
1  class Conveyor {
2  private:
3  size_t elementsCount;
4  size_t queuesCount;
5  size_t averegeTime;
6  const size_t delayTime = 3;
7
8  size_t getCurTime() {
9  return std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::steady_clock::now().
       time_since_epoch()).count();
10 }
11
12 void doObjectLinearWork(matrixObject& curObject, size_t queueNum) {
13 size_t start = getCurTime();
14 // cout << "Object #" << curObject.number << " from Queue #" << queueNum << ":
       START - " << start << endl;
15
16 curObject.addUpMatrix(0, curObject.sizeMatrix/3);
17
18 size_t end = getCurTime();
```

```
19 // cout << "Object #" << curObject.number << " from Queue #" << queueNum << ": STOP
       - " << end << endl;
20 // cout << "Object #" << curObject.number << " from Queue #" << queueNum << ": TIME
       - " << end - start << endl;
21 }
22
23 void doObjectLinearWork2(matrixObject& curObject, size_t queueNum) {
24 size_t start = getCurTime();
25 // cout << "Object #" << curObject.number << " from Queue #" << queueNum << ":
      START - " << start << endl;
26
27 curObject.addUpMatrix(curObject.sizeMatrix / 3, 2 * curObject.sizeMatrix / 3);
28
29 size_t end = getCurTime();
30 // cout << "Object #" << curObject.number << " from Queue #" << queueNum << ": STOP
       - " << end << endl;
31 // cout << "Object #" << curObject.number << " from Queue #" << queueNum << ": TIME
       - " << end - start << endl;
32 }
33
34 void doObjectLinearWork3(matrixObject& curObject, size_t queueNum) {
35 size_t start = getCurTime();
36 // cout << "Object #" << curObject.number << " from Queue #" << queueNum << ":
      START - " << start << endl;
37
38 curObject.addUpMatrix(2 * curObject.sizeMatrix / 3, curObject.sizeMatrix);
39
40 size_t end = getCurTime();
41 // cout << "Object #" << curObject.number << " from Queue #" << queueNum << ": STOP
       - " << end << endl;
42 // cout << "Object #" << curObject.number << " from Queue #" << queueNum << ": TIME
       - " << end - start << endl;
43 //resTimeFile
44 }
45
46 public:
47 Conveyor(size_t elementsCount, size_t queuesCount, size_t milliseconds) : elementsCount(
      elementsCount), queuesCount(queuesCount), averegeTime(milliseconds) {}
48
49 void executeLinear() {
50
51 queue <matrixObject> objectsGenerator;
52
53 for (size_t i = 0; i < elementsCount; ++i) {
54 objectsGenerator.push(matrixObject(1038, -20, 200, i + 1));
55 }
56
57 vector <matrixObject> objectsPool;
58
59 while (objectsPool.size() != elementsCount) {
60 matrixObject curObject = objectsGenerator.front();
61 objectsGenerator.pop();
62
```

```
63 for (size_t i = 0; i < queuesCount; ++i) {
64 if (i == 0) {
65 doObjectLinearWork(curObject, i);
66 } else if (i == 1) {
67 doObjectLinearWork2(curObject, i);
68 } else if (i >= 2) {
69 doObjectLinearWork3(curObject, i);
70 }
71
72 }
73
74 objectsPool.push_back(curObject);
75 }
76 }
77
78 private:
79 void doObjectParallelWork(matrixObject curObject, queue <matrixObject>& queue, size_t
       queueNum, mutex& mutex) {
80 size_t start = getCurTime();
81
82 curObject.addUpMatrix(0, curObject.sizeMatrix/3);
83
84 mutex.lock();
85 queue.push(curObject);
86 mutex.unlock();
87
88 size_t end = getCurTime();
89 // cout << "Object" << curObject.number << "; Queue " << queueNum << "; Time " <<
       end - start << endl;
90 objectTimeStayingAtQueue[queueNum + 1].push_back(-end);
91 }
92
93 void doObjectParallelWork1(matrixObject curObject, queue <matrixObject>& queue, size_t
       queueNum, mutex& mutex) {
94 size_t start = getCurTime();
95 curObject.addUpMatrix(curObject.sizeMatrix / 3, 2 * curObject.sizeMatrix / 3);
96
97 mutex.lock();
98 queue.push(curObject);
99 mutex.unlock();
100
101 size_t end = getCurTime();
102 // cout << "Object" << curObject.number << "; Queue " << queueNum << "; Time " <<
       end - start << endl;
103 objectTimeStayingAtQueue[queueNum + 1].push_back(-end);
104 }
105
106 void doObjectParallelWork2(matrixObject curObject, queue <matrixObject>& queue, size_t
       queueNum, mutex& mutex) {
107 size_t start = getCurTime();
108
109 curObject.addUpMatrix(2 * curObject.sizeMatrix / 3, curObject.sizeMatrix);
110
```

```
111 mutex.lock();
112 queue.push(curObject);
113 mutex.unlock();
114
115 size_t end = getCurTime();
116 // cout << "Object" << curObject.number << "; Queue " << queueNum << "; Time " <<
        end - start << endl;
117 objectTimeStayingAtQueue[queueNum + 1].push_back(-end);
118 }
119
120 public:
121 void executeParallel() {
122
123 queue <matrixObject> objectsGenerator;
124
125 for (size_t i = 0; i < elementsCount; ++i) {
126 objectsGenerator.push(matrixObject(1038, -20, 200, i + 1));
127 }
128
129 vector <thread> threads(3);
130 vector <queue <matrixObject> > queues(3);
131 queue <matrixObject> objectsPool;
132 vector <mutex> mutexes(4);
133 size_t prevTime = getCurTime() - delayTime;
134
135 while (objectsPool.size() != elementsCount) {
136 size_t curTime = getCurTime();
137
138 if (!objectsGenerator.empty() && prevTime + delayTime < curTime) {
139 matrixObject curObject = objectsGenerator.front();
140 objectsGenerator.pop();
141 queues[0].push(curObject);
142
143 prevTime = getCurTime();
144
145 objectTimeStayingAtQueue[0].push_back(-prevTime);
146 }
147
148 for (int i = 0; i < queuesCount; ++i) {
149 if (threads[i].joinable()) {
150 threads[i].join();
151 }
152 if (!queues[i].empty() && !threads[i].joinable()) {
153 mutexes[i].lock();
154 matrixObject curObject = queues[i].front();
155 queues[i].pop();
156 mutexes[i].unlock();
157
158 size_t start = getCurTime();
159 objectTimeStayingAtQueue[i][objectTimeStayingAtQueue[i].size() - 1] += start;
160
161 if (i == 0) {
162 threads[i] = thread(&Conveyor::doObjectParallelWork, this, curObject, ref(queues[i + 1]), i, ref(
```

```
        mutexes[i + 1]));
163 } else if (i == 1) {
164 threads[i] = thread(&Conveyor::doObjectParallelWork1, this, curObject, ref(queues[i + 1]), i, ref(
        mutexes[i + 1]));
165 } else if (i == queuesCount - 1) {
166 threads[i] = thread(&Conveyor::doObjectParallelWork2, this, curObject, ref(objectsPool), i, ref(
        mutexes[i + 1]));
167
168 }
169 }
170 }
171 }
172
173 for (int i = 0; i < queuesCount; ++i) {
174 if (threads[i].joinable()) {
175 threads[i].join();
176 }
177 }
178 }
179
180 };
181
182 int main(int argc, const char * argv[]) { // 1038
183
184 int elementsCount = 100;
185
186 Conveyor conveyor(elementsCount, 3, 5);
187
188 auto start = std::chrono::steady_clock::now();
189
190 conveyor.executeParallel();
191 // conveyor.executeLinear();
192
193 auto end = std::chrono::steady_clock::now();
194 auto duration = std::chrono::duration_cast<std::chrono::milliseconds> (end - start);
195 cout << "\nPROGRAMM ENDED!\n";
196
197 return 0;
198 }
```

2.4 Вывод

В данном разделе была рассмотрена конкретные реализации линейной и конвейерной обработки сложения матриц, необходимые для сравнительного анализа данных реализаций.

3. Экспериментальный раздел

В данном разделе приведены результаты работы двух различных реализаций обработки сложения матриц.

3.1 Сравнительный анализ

Все замеры проводились на процессоре 2.3 GHz Intel Core i5 с памятью 12 ГБ. В таблице 3.1 и на графике 3.1 представлены результаты измерения времени работы линейной и конвейерной реализации обработки сложения матриц.

Таблица 3.1: Время работы различных методов обработки в миллисекундах

Количество объектов	Линейная обработка	Конвейерная обработка
50	1499	1522
100	2896	3025
200	6455	6047
300	12236	9242
400	16805	13934
500	22768	18497
600	26723	22460
700	35227	28188
800	45388	34728
900	59026	42102
1000	68211	49761

Сравнение времени работы приведены для сложения квадратных матриц размера 1038x1038. Такая размерность матрицы была выбрана, из-за того, что реализация линейной и конвейерной обработки основывается на трех очередях, и чтобы загрузить каждую очередь одинаково, нужно выбрать размерность матрицы кратную трем. В нашем случае каждому этапу обработки достается сложение 346 элементов.

3.2 Вывод

По данным эксперимента можно сделать вывод о том, что линейная обработка оказалась менеее эффективной, чем конвейерная. На небольшом количестве объектов эффективность конвейерной обработки не заметна. Это связано с тем, что значительную часть времени работы программы конвейерной обработки занимает инициализация потоков. Но на больших объемах входных данных(1000 обрабатываемых объектов) линейная обработка работает в 1.37 раза дольше.

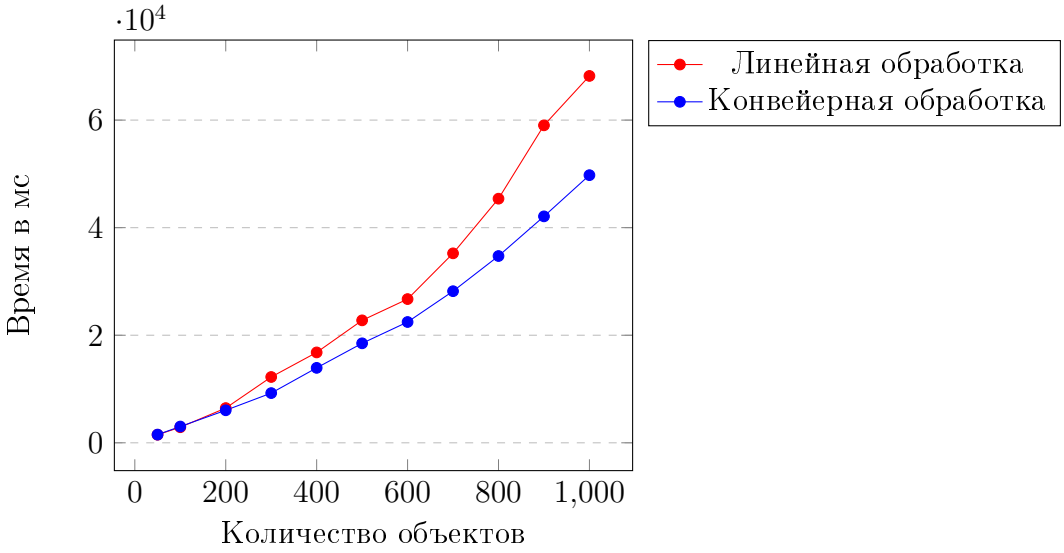

Рис. 3.1: График времени работы различных методов обработки в миллисекундаха

Заключение

В ходе выполнения данной лабораторной работы были изучены принципы конвейерной обработки. Было проведено исследование работы алгоритма при различных параметрах, показавшее, что конвейерная обработка работает значительно быстрее, чем линейная обработка (в 1.37 раза быстрее при количестве объектов, равном 1000).

Литература

[1] ISO/IEC JTC1 SC22 WG21 N 3690 «Programming Languages — C++» [Электронный ресурс]. https://devdocs.io/cpp/

[2] https://cppreference.com/ [Электронный ресурс]