МГТУ им. Баумана

Лабораторная работа №5

По курсу: "Анализ алгоритмов"

Конвейер

Работу выполнила: Серёгина Дарья, ИУ7-54Б

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

Москва, 2020

Оглавление

Введение **2**

1 Аналитическая часть **3**
1.1 Общие сведения о конвейерной обработке 3
1.2 Параллельное программирование 3
1.2.1 Организация взаимодействия параллельных потоков 5
1.3 Вывод 5

2 Конструкторская часть **6**
2.1 Организация обработки 6
2.2 Вывод 6

3 Технологическая часть **7**
3.1 Выбор ЯП 7
3.2 Листинг кода алгоритмов 7
3.3 Вывод 13

4 Исследовательская часть **14**
4.1 Сравнительный анализ на основе замеров времени 14
4.2 Тестирование 15
4.3 Вывод 15

Заключение **16**

Список литературы **16**

Введение

Цель работы: создать систему конвейерной обработки.

Задачи данной лабораторной работы:

1. спроектировать ПО, реализующего конвейерную обработку;
2. описать реализацию ПО;
3. провести тестирование ПО.

1 Аналитическая часть

В данной части будут рассмотрены главные принципы конвейерной обработки и параллельных вычислений.

1.1 Общие сведения о конвейерной обработке

Конвейер – машина непрерывного транспорта [1], предназначенная для перемещения сыпучих, кусковых или штучных грузов.
Конвейерное производство - система поточной организации производства на основе конвейера, при которой оно разделено на простейшие короткие операции, а перемещение деталей осуществляется автоматически. Это такая организация выполнения операций над объектами, при которой весь процесс воздействия разделяется на последовательность стадий с целью повышения производительности путём одновременного независимого выполнения операций над несколькими объектами, проходящими различные стадии. Конвейером также называют средство продвижения объектов между стадиями при такой организации[2]. Появилось в 1914 году на производстве Модели-Т на заводе Генри Форда[3] и произвело революцию сначала в автомобилестроении, а потом и во всей промышленности.

1.2 Параллельное программирование

Параллельные вычисления — способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (од-

новременно).

При использовании многопроцессорных вычислительных систем с общей памятью обычно предполагается, что имеющиеся в составе системы процессоры обладают равной производительностью, являются равноправными при доступе к общей памяти, и время доступа к памяти является одинаковым (при одновременном доступе нескольких процессоров к одному и тому же элементу памяти очередность и синхронизация доступа обеспечивается на аппаратном уровне). Многопроцессорные системы подобного типа обычно именуются симметричными мультипроцессорами (symmetric multiprocessors, SMP).

Перечисленному выше набору предположений удовлетворяют также активно развиваемые в последнее время многоядерные процессоры, в которых каждое ядро представляет практически независимо функциони рующее вычислительное устройство.

Обычный подход при организации вычислений для многопроцессорных вычислительных систем с общей памятью – создание новых параллельных методов на основе обычных последовательных программ, в которых или автоматически компилятором, или непосредственно программистом выделяются участки независимых друг от друга вычислений. Возможности автоматического анализа программ для порождения параллельных вычислений достаточно ограничены, и второй подход является преобладающим. При этом для разработки параллельных программ могут применяться как новые алгоритмические языки, ориентированные на параллельное программирование, так и уже имеющиеся языки, расширенные некоторым набором операторов для параллельных вычислений.

Широко используемый подход состоит и в применении тех или иных библиотек, обеспечивающих определенный программный интерфейс (application programming interface, API) для разработки параллельных программ. В рамках такого подхода наиболее известны Windows Thread API. Однако первый способ применим только для ОС семейства Microsoft Windows, а второй вариант API является достаточно трудоемким для использования и имеет низкоуровневый характер [8].

1.2.1 Организация взаимодействия параллельных потоков

Потоки исполняются в общем адресном пространстве параллельной программы. Как результат, взаимодействие параллельных потоков можно организовать через использование общих данных, являющихся доступными для всех потоков. Наиболее простая ситуация состоит в использовании общих данных только для чтения. В случае же, когда общие данные могут изменяться несколькими потоками, необходимы специальные усилия для организации правильного взаимодействия.

1.3 Вывод

В данном разделе были рассмотрены основы конвейерной обработки, технология параллельного программирования и организация взаимодействия параллельных потоков.

2 | Конструкторская часть

Требования к вводу: Количество конвейеров должно быть больше 0.
Требования к программе при параллельной обработке:

- Объекты должны последовательно проходить конвейеры в заданном подядке;
- конвейеры должны работать каждый в своем потоке;
- конвейер должен завершать свою работу при поступлении специального элемента;
- до завершения работы конвейер должен ожидать поступления новых элементов.

2.1 Организация обработки

У каждой линии конвейера есть очередь элементов. Когда линия еще активна, но элементов в очереди нет, линия уходит в режим ожидания. По прошествию заданного времени линия проверяет не появились ли новые элементы в очереди. Если очередь не пустая, то нужно получить и обработать элемент, передать его следующей линии, если такая существует.

2.2 Вывод

В данном разделе была рассмотрена схема организации конвейерной обработки.

3 | Технологическая часть

Замеры времени были произведены на: Intel(R) Core(TM) i5-8300H, 4 ядра, 8 логических процессоров.

3.1 Выбор ЯП

В качестве языка программирования был выбран C++ [?], так как этот язык поддерживает управление потоками на уровне ОС. Средой разработки Visual Studio. Для измерения процессорного времени была взята функция rdtsc из библиотеки ctime.

3.2 Листинг кода алгоритмов

Листинг 3.1: Код программы

```
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 };
```

3.3 Вывод

В данном разделе были рассмотрены основные сведения о модулях программы, листинг кода.

4 Исследовательская часть

4.1 Сравнительный анализ на основе замеров времени

Был проведен замер времени работы конвейерной и линейной обработки при разных временах обработки одной линии.

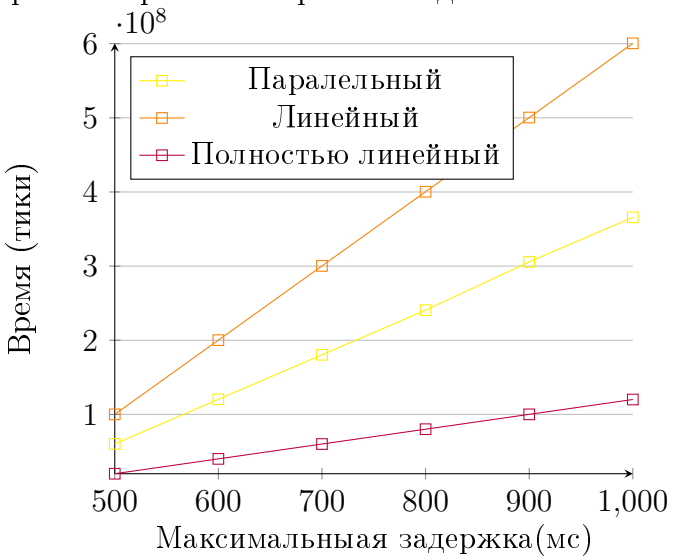

Рис. 4.1: Сравнение времени работы при разных дельтах зедержек

На графиках видно, что конвейерная обработка с параллельными потоками в 2.5 раза быстрее чем такая же линейная с захватом переменных. Если же убрать ненужные захваты переменных (т.к. все действия происходят линейно, не может быть одновременного обращения к одной области памяти), то это будет работать в 2.5 раза быстрее, чем параллельный.

4.2 Тестирование

Для тестирования был выведен лог операций в формате : id конвейера; id элемента; состояние; время

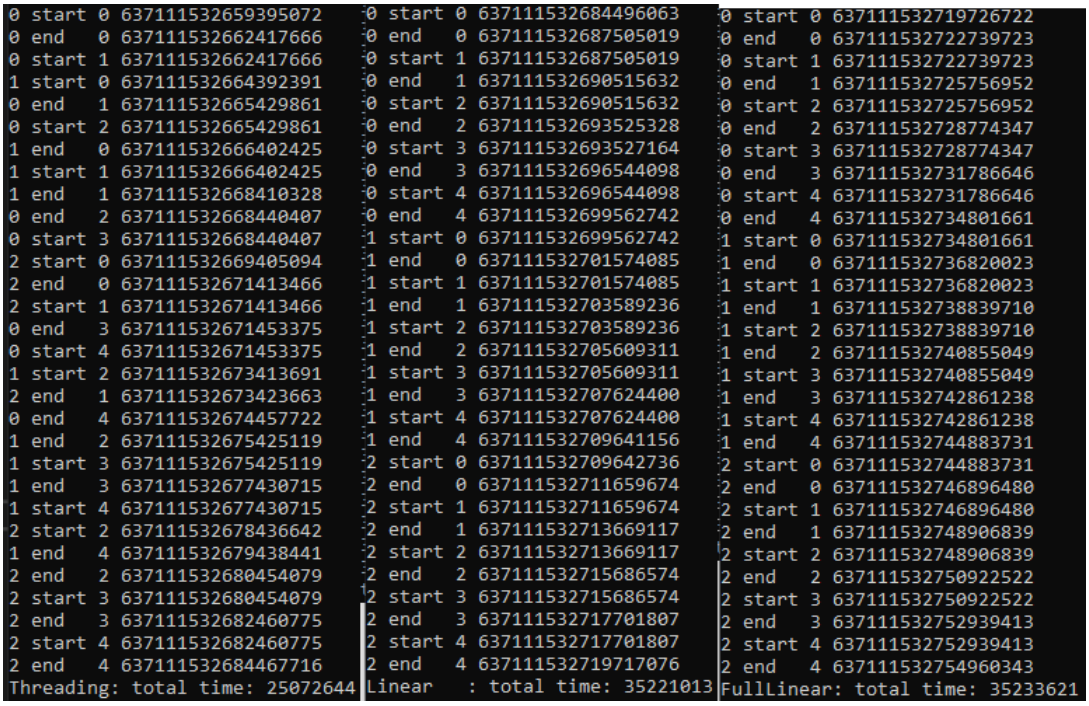

Рис. 4.1: Лог работы конвейерной обработки

4.3 Вывод

По результатам исследования конвейерную обработку нет смысла применять для задач, занимающих мало времени, т.к. в этом случее большая часть времени потратится на ожидание доступа к переменной, дополнительных проверок. Тестирование показало, что конвейерная обработка реализована правильно.

Заключение

В ходе лабораторной работы я изучила возможности применения параллельных вычислений и конвейерной обработки и использовала такой подход на практике.

Был проведен эксперимент с разными дельтами задержек, который показал что если первый конвейер тормозит работу, то общее время работы системы линейно от задержки первого конвейера. Также этот эксперимент показал, что конвейерную обработку нет смысла применять для задач, занимающих мало времени, т.к. в этом случее большая часть времени потратится на ожидание доступа к переменной, дополнительных проверок.

Конвейерная обработка позволяет сильно ускорить программу, если требуется обработать набор из однотипных данных, причем алгоритм обработки должен быть разбиваем на стадии. Однако от конвейерной обработки не будет смысла, если одна из стадий намного более трудоемкая, чем остальные, так как производительность всей программы будет упираться в производительность этой самой стадии, и разницы между обычной обработкой и конвейерной не будет, только добавятся накладные вычисления, связанные с диспетчеризацией потоков. В таком случае можно либо разбить трудоемкую стадию на набор менее трудоемких, либо выбрать другой алгоритм, либо отказаться от конвейерной обработки.

Литература

[1] Меднов В.П., Бондаренко Е.П. Транспортные, распределительные и рабочие конвейеры. М., 1970.

[2] Конвейерное производство[Электронный ресурс] - режим доступа https:/ё/dic.academic.ru/dic.nsf/ruwiki/1526795

[3] Конвейерный метод производства Генри Форда[Электронный ресурс] - режим доступа https://popecon.ru/305-konveiernyi-metod-proizvodstva-genri-forda.html

[4] И. В. Белоусов(2006), Матрицы и определители, учебное пособие по линейной алгебре, с. 1 - 16

[5] Константин Баркалов, Владимир Воеводин, Виктор Гергель. Intel Parallel Programming [Электронный ресурс], - режим доступа https://www.intuit.ru/studies/courses/4447/983/lecture/14925

[6] И. В. Белоусов(2006), Матрицы и определители, учебное пособие по линейной алгебре, с. 1 - 16

[7] Le Gall, F. (2012), "Faster algorithms for rectangular matrix multiplication Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2012), pp. 514–523

[8] Константин Баркалов, Владимир Воеводин, Виктор Гергель. Intel Parallel Programming [Электронный ресурс], - режим доступа https://www.intuit.ru/studies/courses/4447/983/lecture/14925