Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Отчет о лабораторной работе

Курс: Параллельные вычисления
Тема: Создание многопоточных программ на языке C++ с использонием Pthreads и OpenMP

Выполнила студентка группы 13541/3 _____ Мельникова Д.Н.

Преподаватель _____ Стручков И.В.

Санкт-Петербург
2018

Содержание

1 Постановка задачи

1.1 Индивидуальное задание

Вариант 5, OpenMP.

Поиск кратчайшего пути в ориентированном графе (алгоритм Беллмана-Форда).

1.2 Программа работы

1. Для алгоритма из полученного задания написать последовательную программу

на языке C или C++, реализующую этот алгоритм.

2. Для созданной последовательной программы необходимо написать 3-5 тестов,

которые покрывают основные варианты функционирования программы. Для создания тестов можно воспользоваться механизмом Unit-тестов среды NetBeans,

или описать входные тестовые данные в файлах. При использовании NetBeans

необходимо в свойствах проекта установить ключ компилятора -pthread.

3. Проанализировать полученный алгоритм, выделить части, которые могут быть

распараллелены, разработать структуру параллельной программы. Определить

количество используемых потоков, а также правила и используемые объекты синхронизации.

4. Согласовать разработанную структуру и детали реализации параллельной программы с преподавателем.

5. Написать код параллельной программы и проверить ее корректность на созданном ранее наборе тестов. При необходимости найти и исправить ошибки.

6. Провести эксперименты для оценки времени выполнения последовательной и параллельной программ. Проанализировать полученные результаты.

7. Сделать общие выводы по результатам проделанной работы:

- Различия между способами проектирования последовательной и параллельной реализаций алгоритма.
- Возможные способы выделения параллельно выполняющихся частей, возможные правила синхронизации потоков
- Сравнение времени выполнения последовательной и параллельной программ.
- Принципиальные ограничения повышения эффективности параллельной реализации по сравнению с последовательной

2 Сведения о системе

Linux Mint-MATE 4.10.0-38-generic #42~16.04.1-Ubuntu SMP T x86_64 GNU/Linux

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 94
Model name:            Intel(R) Core(TM) i5-6600 CPU @ 3.30GHz
Stepping:              3
CPU MHz:               800.024
CPU max MHz:           3900,0000
CPU min MHz:           800,0000
BogoMIPS:              6624.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):     0-3
```

3 Структура проекта

```
.
├── bin
│   ├── openmpBF
│   ├── pthreadBF
│   └── strictBF
├── build.sh
├── graph.txt
├── mkGraph
├── mkGraph.hs
├── README.md
├── run.sh
└── src
    ├── main.cpp
    └── pt_main.cpp
```

Build.sh собирает файл генерации графа и бинарные файлы openmpBF, strictBF и pthreadBF:

ghc mkGraph.hs

./mkGraph $1

mkdir bin

g++ -std=c++11 -o bin/strictBF src/main.cpp

g++ -std=c++11 -fopenmp -o bin/openmpBF src/main.cpp

g++ -std=c++11 -pthread -o bin/pthreadBF src/pt_main.cpp

Run.sh запускает все три программы на сгенерированном графе:

```
echo "Simple running: "
./bin/strictBF graph.txt
echo "================="
echo "OpenMP running: "
./bin/openmpBF graph.txt
echo "================="
echo "pthreads running: "
./bin/pthreadBF graph.txt
echo "================="
```

4 Алгоритм решения

Реализованный алгоритм Беллмана-Форда выглядит следующим образом:

```
for (int i = 0; i < v_cnt - 1; i++)
  {
    for (int j = 0; j < e_cnt; j++)
      if (d[ gr[j].start ] < imax)
        {
          if (d[ gr[j].end ] > d[ gr[j].start ] + gr[j].weight)
            {
              d[ gr[j].end ] = d[ gr[j].start ] + gr[j].weight;
              p[ gr[j].end ] = gr[j].start;
            }
        }
  }
```

, где gr – вектор, содержащий граф в виде списка ребер, d – вектор, содержащий путь от стартовой вершины до всех остальных, p – это вектор для восстановления минимального пути.

Граф считывается из текстового файла вида:

v_cnt e_cnt start_v end_v

[st_v e_v weight] * e_cnt,

где v_cnt – количество вершин, e_cnt – количество ребер, start_v и end_v – начало и конец искомого пути, далее следуют e_cnt описания ребер вида: начальная вершина – конечная вершина – вес ребра.

Такой граф генерируется программой mkGraph, реализованной на языке Haskell. Для ее сборки потребуется компиллятор ghc. Она принимает один агрумент – random seed для числа вершин. Далее случайно выбирается количество ребер из промежутка [v_cnt, v_cnt^2], также случайно выбираются начальная и конечная вершины пути и генерируется необходимое число ребер.

5 Распараллеливание алгоритма

OpenMP

Как видно выше, алгоритм состоит из двух вложенных циклов, каждый из которых может быть распараллелен. Значение каждой итерации независимо от предыдущих вычислений.

Тогда, достаточно проставить директиву #pragma omp parallel for над циклом для распараллеливания:

```
 for (int i = 0; i < v_cnt - 1; i++)
     {
       #pragma omp parallel num_threads(NUM_THREADS)
       for (int j = 0; j < e_cnt; j++)
          if (d[ gr[j].start ] < imax)
            {
              if (d[ gr[j].end ] > d[ gr[j].start ] + gr[j].weight)
                {
                  d[ gr[j].end ] = d[ gr[j].start ] + gr[j].weight;
                  p[ gr[j].end ] = gr[j].start;
                }
            }
     }
```

Pthreads

Распараллеливание посредством posix threads реализовано в файле src/pt_main.cpp.

В данном случае, функция, выполняющаяся для каждого потока выносится отдельно (thread), в нее передается указатель типа void на аргументы, который в поледствии приводится к нужному типу.

В начале функции bf выполнение разделяется на потоки в необходимом количестве (1, 2, 4, 8). В каждый поток передается метаинформация, разделяемая между потоками – упомянутые выше массивы gr, d и p. Это делается для того, чтобы не переполнять стек – для тестирования на больших графах.

6 Тестирование

Для подсчета статистики был написан скрипт test.sh:

```
for (( i=1; i<=50; i++))
    do
        ./bin/openmpBF graph.txt >> openmp1.txt
        ./bin/pthreadBF graph.txt >> pthread1.txt
      echo "Number $i"
    done
```

Такой скрипт по 50 раз запускает программы с OpenMP и Pthreads и распределяет их результат по файлам.

Проведем такие эксперименты для 1, 2, 4 и 8 параллельных потоков исполнения и соберем результаты в csv файл, рассчитаем статистику (в представленном эксперименте 345 вершин и 64764 ребра). Полные статистические данные представленны в Приложениях.

	1		2		4		8	
	openMP	pthreads	openMP	pthreads	openMP	pthreads	openMP	pthreads
avg	1.1019924	1.168518	0.59684358	0.64566532	0.57498054	0.60760008	0.60732348	0.59292478
disperse	0.004887755	0.061882824	0.008365093	0.116122141	0.084770861	0.075596543	0.017693692	0.041166406
матожидние	0.001354792	0.0171527285	0.0023186429	0.0321868239	0.0234968521	0.0209539076	0.0049043511	0.0114105359

Выводы

OpenMP быстрее последовательного алгоритма в 1.847 раз, pthread – в 1.810, как показали эксперименты на 2-х ядерной машине.

Как показывает статистика, наиболее быстрым решением являются POSIX threads: у них в среднем минимальные средние временные затраты на подсчет и дисперсия.