Санкт-Петербургский политехнический университет Петра Великого

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Кафедра компьютерных систем и программных технологий

Параллельные вычисления

Отчет по курсовой работе

Создание многопоточных программ на языке C++ с использованием OpenMP

Работу выполнила:

Михалёва М.В.

Группа:
3540901/91502

Преподаватель:

Стручков И.В.

Санкт-Петербург
2020 г.

Оглавление

1. Цель работы

Вариант 5. OpenMP. Поиск кратчайшего пути в ориентированном графе (алгоритм БФ).

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.
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 параллельно 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 программы. Проанализировать полученные результаты.
7. Сделать общие выводы по результатам проделанной работы

3. Характеристики системы

```
rheon@rheon-VirtualBox:~$ uname -a
Linux rheon-VirtualBox 4.15.0-45-generic #48~16.04.1-Ubuntu SMP Tue Jan 29 18:03
:48 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
```

```
rheon@rheon-VirtualBox:~$ hwinfo --short
cpu:
                        Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1800 MHz
                        Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1800 MHz
                        Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1800 MHz
                        Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1800 MHz
```

4. Ход работы

4.1 Структура проекта

Проект содержит следующие основные файлы:

- Bf-omp.h, bf-omp.cpp – параллельная реализация
- Bp-single.h, bf-single.cpp – последовательная реализация
- Main.cpp
- Util.h, util.cpp – вспомогательные функции
- Genmat.cpp – генерация случайной матрицы. Позволяет указывать количество узлов, вероятность существования ребра между двумя узлами, среднее значение веса положительного цикла, среднее значение веса отрицательного цикла, вероятность существования отрицательного цикла.

4.2 Алгоритм БФ

Алгоритм Беллмана-Форда – алгоритм поиска кратчайшего пути в графе. За время O(|V| x |E|) алгоритм находит кратчайшие пути от одной вершины графа до всех остальных. В отличие от алгоритма Дейкстры, алгоритм Беллмана-Форда допускает рёбра с отрицательным весом.

Входные данные: Граф и начальная вершина src.
Выходные данные: Кратчайшее расстояние до всех вершин от src. Если попадается цикл отрицательного веса, то самые короткие расстояния не вычисляются, выводится сообщение о наличии такого цикла.

1. На этом шаге инициализируются расстояния от исходной вершины до всех остальных вершин, как бесконечные, а расстояние до самого src принимается равным 0. Создается массив dist[] размера |V| со всеми значениями равными бесконечности, за исключением элемента dist[src], где src — исходная вершина.
2. Вторым шагом вычисляются самые короткие расстояния. Следующие шаги нужно выполнять |V|-1 раз, где |V| — число вершин в данном графе.
    - Произведите следующее действие для каждого ребра u-v: Если dist[v] > dist[u] + вес ребра uv, то обновить dist[v] dist [v] = dist [u] + вес ребра uv
3. На этом шаге сообщается, присутствует ли в графе цикл отрицательного веса. Для каждого ребра u-v необходимо выполнить следующее:

    - Если dist[v] > dist[u] + вес ребра uv, то в графе присутствует цикл отрицательного веса.
      Идея шага 3 заключается в том, что шаг 2 гарантирует кратчайшее расстояние, если граф не содержит цикла отрицательного веса. Если мы снова переберем все ребра и получим более короткий путь для любой из вершин, это будет сигналом присутствия цикла отрицательного веса.

4.3 Реализация последовательной программы

В листинге 1 приведена программа, отображающая последовательный алгоритм.

Листинг 1.

```
#include "bf-omp.h"
#include <algorithm>
#include "util.h"

#include <iostream>

using namespace std;

void bf_single(int p, int n, int *mat, int *dist, bool
*has_negative_cycle) {
    (void) p;
    dist[0] = 0;
    for (int i = 1; i < n; ++i)
        dist[i] = INF;
    bool has_change = false;

    bool * last_round = new bool[n];
    fill_n(last_round, n, false);
    last_round[0] = true;

    bool * this_round = new bool[n];
```

```
    fill_n(this_round, n, false);

    int *times = new int[n];
    fill_n(times, n, 0);

    int my_load = n;
    int my_begin = 0;
    int my_end = my_begin + my_load;
    while (true) {
        // 1 round
        has_change = false;
        for (int u = 0; u < n; u++) {
            if (last_round[u]) {
                for (int v = my_begin; v < my_end; ++v) {
                    int weight = mat[u * n + v];
                    if (weight < INF)
                        if (dist[u] + weight < dist[v]) {
                            dist[v] = dist[u] + weight;
                            times[v] += 1;
                            this_round[v] = true;
                            has_change = true;
                            if (v == 0 && dist[v] < 0) {
                                *has_negative_cycle = true;
                            }
                            if (times[v] == n) {
                                *has_negative_cycle = true;
                            }
                        }
                }
            }
        }
        if (!has_change) {
            goto END;
        }
        if (*has_negative_cycle) {
            goto END;
        }
        swap(last_round, this_round);
        fill_n(this_round, n, false);
    }
    END : {}


    delete[] last_round;
    delete[] this_round;
    delete[] times;
}
```

Проект, со всеми исходным данными можно найти в репозитории (https://github.com/mariarheon/parallel-Bellman-Ford).

На вход функции `bf_single` подается количество потоков, количество узлов, матрица, дистанция, а так же метка наличия отрицательного цикла. Затем осуществляется поиск путей, в соответствии с описанным ранее алгоритмом. В ходе выполнения изменяются значения посчитанной дистанции и метки `наличия отрицательного цикла.`

4.4 Реализация параллельного алгоритма с использованием OpenMP

Структура кода очень похожа на последовательную версию программы. Основной массив прохода по узлам графа остался прежним.

Директива #pragma omp parallel указывает на то, что данный цикл следует разделить по итерациям между потоками. num_threads(p) указывает на количество потоков, обрабатывающих данные участок. В начале данного участка были инициализированы переменные, уникальные для каждого потока, такие как номер потока, начальный и конечный узлы.

Директива #pragma omp barrier позволяет синхронизировать код таким образом, что дальнейший код программы не будет выполнен до тех пор, пока все потоки не достигнут данного барьера. Как только барьер достигнут всеми потоками, выполнение программы продолжается.

С помощью директивы #pragma omp critical был выделен участок кода, который будет исполняться только одним потоком в один момент времени. Если данная критическая секция уже выполняется каким-либо потоком, то все другие потоки, выполнившие директиву для данной секции, заблокированы, пока исполняющий поток не закончит выполнение данной критической секции.

В листинге 2 приведена программа, отображающая параллельный алгоритм

Листинг 2.

```
#include "bf-omp.h"
#include "omp.h"
#include <algorithm>
#include "util.h"

#include <iostream>

using namespace std;

void bf_omp(int p, int n, int *mat, int *dist, bool
*has_negative_cycle) {
    // allocation
    int q = n / p, r = n % p;
    int load[p], begin[p];
    load[0] = q;
    for (int i = 1; i < p; ++i)
        load[i] = q + ((i <= r) ? 1 : 0);
    begin[0] = 0;
    for (int i = 1; i < p; ++i)
        begin[i] = begin[i - 1] + load[i - 1];

    // initialization
    dist[0] = 0;
    for (int i = 1; i < n; ++i)
        dist[i] = INF;
```

```
    bool has_change = false;
    *has_negative_cycle = false;

    bool *last_round = new bool[n];
    fill_n(last_round, n, false);
    last_round[0] = true;

    bool *this_round = new bool[n];
    fill_n(this_round, n, false);

    int *times = new int[n];
    fill_n(times, n, 0);
    #pragma omp parallel num_threads(p)
    {
        int my_rank = omp_get_thread_num();
        int my_load = load[my_rank];
        int my_begin = begin[my_rank];
        int my_end = my_begin + my_load;
        bool my_has_change = false;

        #pragma omp barrier

        for (size_t i = 0; ; ++i) {
            has_change = my_has_change = false;
            for (int u = 0; u < n; u++) {
                if (last_round[u]) {
                    for (int v = my_begin; v < my_end; ++v) {
                        int weight = mat[u * n + v];
                        if (weight < INF)
                            if (dist[u] + weight < dist[v]) {
                                #pragma omp critical
                                {
                                    dist[v] = dist[u] + weight;
                                    times[v] += 1;
                                }
                                this_round[v] = true;
                                my_has_change = true;
                                if (v == 0 && dist[v] < 0) {
                                    *has_negative_cycle = true;
                                }
                                if (times[v] == n) {
                                    *has_negative_cycle = true;
                                }
                            }
                    }
                }
            }
            #pragma omp barrier
            #pragma omp critical
            {
```

```
                has_change = has_change || my_has_change;
            }
            #pragma omp barrier
            if (!has_change) {
                goto END;
            }
            if (*has_negative_cycle) {
                goto END;
            }
            #pragma omp barrier
            if (my_rank == 0) {
                swap(last_round, this_round);
                fill_n(this_round, n, false);
            }
            #pragma omp barrier
        }
        END : {}
    }

    delete[] last_round;
    delete[] this_round;
    delete[] times;
}
```

5. Тестирование

Для тестирования была создана отдельная программа test.cpp.

Программа была запущена 50 раз, с числом потоков = 4, вероятностью существования ребра между двумя узлами = 0.5 и вероятностью наличия отрицательного веса = 0. Результаты приведены в с.

Получили следующие усредненные данные:

Таблица 1. Измерение скорости выполнения

Количество узлов	Средний вес узла	Последовательный алгоритм	Параллельный алгоритм
100	10000	0,000349	0,000357
1000	10000	0,033884	0,012741
10000	10000	4,575381	1,968042

Заметим, что выигрыш в производительности происходит при большем числе узлов.

Далее, выполним тестирование для различного числа потоков. Был выбран граф размером 10000 узлов, средним весом = 10000. Тесты так же запускались 50 раз для каждого случая.

По полученным данным был произведен расчет математического ожидания идисперсии. Доверительный интервал выбран = 99%. Следовательно, коэффициент доверия = 2.6778. Результаты приведены в мс.

Таблица 2. Результаты запусков

Количество потоков	Мат. Ожидание	Дисперсия	Доверительный интервал
1	4965,15	23142,4	4965,15 +- 128,82
2	2588,33	8255,55	2588,33 +- 122,9398
4	1716,97	20071	1716,97 +- 119,967
8	1684,07	8900,36	1684,07 +- 79,8881
16	1814,74	22466,3	1814,74 +-126,924
32	1854,43	21770,6	1854,43 +- 153,895

6. Выводы

В рамках данной работы были изучены основы многопоточных реализация программ на C++ с использованием библиотеки OpenMP. Для написанной программы было проведено тестирование при разном количестве потоков, а также были оценены вероятностные характеристики времени работы.

По результатам было установлено, что многопоточная реализация работает в разы эффективнее последовательной программы для большого количества узлов. При небольших значениях скорость последовательной и параллельной программ приблизительно одинакова. Такой итог может быть обоснован тем, что решаемая задача для маленького числа узлов крайне проста, и накладные расходы на выполнение параллельной программы не обоснованы.

Таким образом, можно сделать вывод, что использование средств OpenMP значительно упрощает написание кода, а также увеличивает эффективность выполнения сложных алгоритмов.