ИИКС

Лабораторная работа №2: «Выделение ресурса параллелизма. Технология OpenMP»

Грущин Илья

Б21-515

2023 г.

Описание используемой рабочей среды

Виртуальная машина: VirtualBox

Модель процессора: 11th Gen Intel(R) Core(TM) i5-11400 @ 2.70GHz, ядер: 4, логических процессоров: 6

Объем оперативной памяти: 6.0 Гб

Тип оперативной памяти: DDR4

Операционная система: Fedora Linux 38 (Workstation Edition)

Разрядность ОС: 64

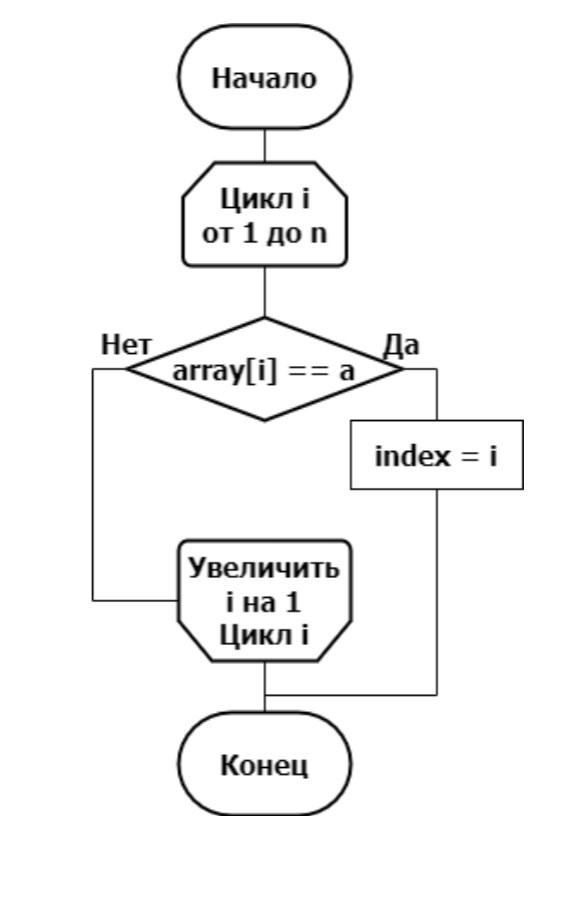
Среда разработки: gcc 13.2.1

Поддерживаемая версия OpenMP: 4.5

Анализ приведенного алгоритма

Описание принципа работы алгоритма: линейный перебор элементов массива и сравнение каждого элемента с числом a, если элемент массива равен a, то сохраняем индекс и выходим из цикла, иначе — переход к следующему элемента..

Блок-схема алгоритма:



Описание используемых директив и функций OpenMP

* #pragma omp parallel для инициализации области, выполняемой несколькими потоками
* #pragma omp for для распределения итераций между потоками
* #pragma omp cancel для остановки цикла при нахождении искомого элемента.
* omp\_get\_wtime() для измерения времени выполнения параллельной области.

Временная сложность алгоритма

Временная сложность последовательного перебора — O(n)

В лучшем случае — понадобится всего 1 операция (если искомое число в начале массива)  
В худшем случае — понадобится n операций (если искомого числа нет в массиве, или оно находится в конце)

Графики

На обоих графиках ось абсцисс — количество потоков.

Заключение

Написана программа для измерения времени работы последовательного и параллельного линейного алгоритма поиска максимального элемента в массиве и подсчета среднего количества операций сравнения на поток. Теоретические и экспериментальные данные не совпадают. Это может быть связано с тем, что на практике тратится время на синхронизацию при обращении к разделяемым переменным, создание и переключение потоков, а также на магическое распределение нагрузки внутри виртуальной машины, поэтому при использовании более 4-х потоков уменьшается ускорение.

Приложение

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <omp.h>

int\*\* makeArr(int rows, int columns) {

    int \*\*result = (int\*\*) malloc(rows \* sizeof(int\*));

    for (int i = 0; i < rows; ++i) {

        result[i] = (int\*) malloc(columns \* sizeof(int));

    }

    return result;

}

void fillRand(int\*\* arrays, int rows, int columns) {

    const int random\_seed = 920215;

    for (int i = 0; i < rows; ++i) {

        srand(random\_seed);

        for (int j = 0; j < columns; ++j) {

            arrays[i][j] = rand();

        }

    }

}

void printAvTime(int\* threadNums, double\* avtime, int threadNumsLen) {

    printf("in milliseconds\n");

    for (int threadId = 0; threadId < threadNumsLen; ++threadId) {

        printf("%d: %.10lf\n", threadNums[threadId], avtime[threadId] \* 1000.0);

    }

}

void printDel(int minsCount) {

    for (int minsId = 0; minsId < minsCount; ++minsId) {

        printf("-");

    }

    printf("\n");

}

void printAvCount(int\* threadNums, int\*\* avCount, int threadNumsLen) {

    char \*fmt = "%-10d";

    int dellen = 10 \* (threadNumsLen + 1);

    printDel(dellen);

    printf("#%-9d", "");

    for (int measId = 0; measId < 10; ++measId) {

        printf(fmt, threadNums[measId]);

    }

    printf("...%d, ...%d, ...%d\n", 16, 32, 64);

    printDel(dellen);

    for (int measId = 0; measId < threadNumsLen; ++measId) {

        printf(fmt, threadNums[measId]);

        for (int arrId = 0; arrId < threadNumsLen; ++arrId) {

            printf(fmt, avCount[measId][arrId]);

        }

        printf("\n");

    }

    printDel(dellen);

}

int main(int argc, char\* argv[]) {

    // Determine the OpenMP support

    printf("OpenMP: %d;\n======\n", \_OPENMP);

    // Init data

    const int arLen = 10000000;

    const int arNum = 10;

    int threadNumsLen = 13;

    int threadsNum[13] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 16, 32, 64};

    // Init measurement arrays

    int \*\*avCount = makeArr(threadNumsLen, 64);

    double \*avTime = (double\*) malloc(threadNumsLen \* sizeof(double));

    // Init arrays

    int \*\*arrays = makeArr(arNum, arLen);

    fillRand(arrays, arNum, arLen);

    // Start measuring

    for (int measId = 0; measId < threadNumsLen; ++measId) {

        int threadCount = threadsNum[measId];

        double totalTime = 0;

        // Measuring time

        for (int arId = 0; arId < arNum; ++arId) {

            int max = -1;

            double startTime = omp\_get\_wtime();

            #pragma omp parallel num\_threads(threadCount) shared(arId, totalTime, arrays, arLen) reduction(max: max) default(none)

            {

                #pragma omp for

                for (int posArr = 0; posArr < arLen; ++posArr) {

                    if (arrays[arId][posArr] > max) {

                        max = arrays[arId][posArr];

                    }

                }

            }

            totalTime += omp\_get\_wtime() - startTime;

        }

        avTime[measId] = totalTime / arNum;

        // Measuring comparations

        for (int arId = 0; arId < arNum; ++arId) {

            int max = -1;

            #pragma omp parallel num\_threads(threadCount) shared(measId, arId, avCount, arrays, arLen) reduction(max: max) default(none)

            {

                #pragma omp for

                for (int posArr = 0; posArr < arLen; ++posArr) {

                    avCount[measId][omp\_get\_thread\_num()] += 1;

                    if (arrays[arId][posArr] > max)

                    {

                        max = arrays[arId][posArr];

                    }

                }

            }

        }

        // Average comparations per thread

        for (int threadId = 0; threadId < threadsNum[measId]; ++threadId) {

            avCount[measId][threadId] /= arNum;

        }

    }

    printAvTime(threadsNum, avTime, threadNumsLen);

    printf("\n");

    printAvCount(threadsNum, avCount, threadNumsLen);

    return 0;

}

Приложение

Таблицы с теоретическими результатами и результатами вычислительных экспериментов Способ вычисления ускорения: T1 – время работы алгоритма с одним потоком (14,66 мс)

Tn – время работы алгоритма с n потоками

Sn – ускорение с n потоками, Sn = T1/Tn

En – эффективность с n потоками, En = Sn/n

Теоретические ускорение и эффективность определены лишь приблизительно. Предположительно ускорение с использованием 1-6 потоков будет равно количеству потоков, а затем начнет падать, т. к. физических ядер меньше, чем используемых потоков. Соответственно в момент времени будет исполняться не более 6 потоков, а вот времени на распределение данных, их синхронизацию и в целом организацию параллельного вычисления будет тратиться больше.

Значения теоретического ускорения определены на глаз исходя из расчета, что использование лишнего потока уменьшит ускорение значительно, но добавление каждого следующего потока будет оказывать все более незначительное влияние.

Теор. Ускорение = теор. Эффективность / количество потоков.

