Национальный исследовательский ядерный университет «МИФИ»
Кафедра №42
«Криптология и кибербезопасность»

Лабораторная работа №2
Выделение ресурса параллелизма. Технология OpenMP

Саменков Андрей
Б20-515

4 октября 2022

1 Описание используемой рабочей среды

1. Процессор: Ryzen 5 5500U 6 cores 12 threads
2. ОЗУ: 8gb DDR4
3. Система: Linux Fedora 36 Workstation x64
4. Компилятор: G++ (GCC) 12.2.1 20220819
5. Версия OpenMP: 4.5(201511)

2 Алгоритм

Создается несколько рандомных массивов. Проходим по каждому элементу массива и сравниваем его с target, и если они равны, то завершаем поиск и все потоки. Таким образом находим target в массиве. Создается некоторое количество потоков. Сложность алгоритма $O(n)$.

3 Блок схема

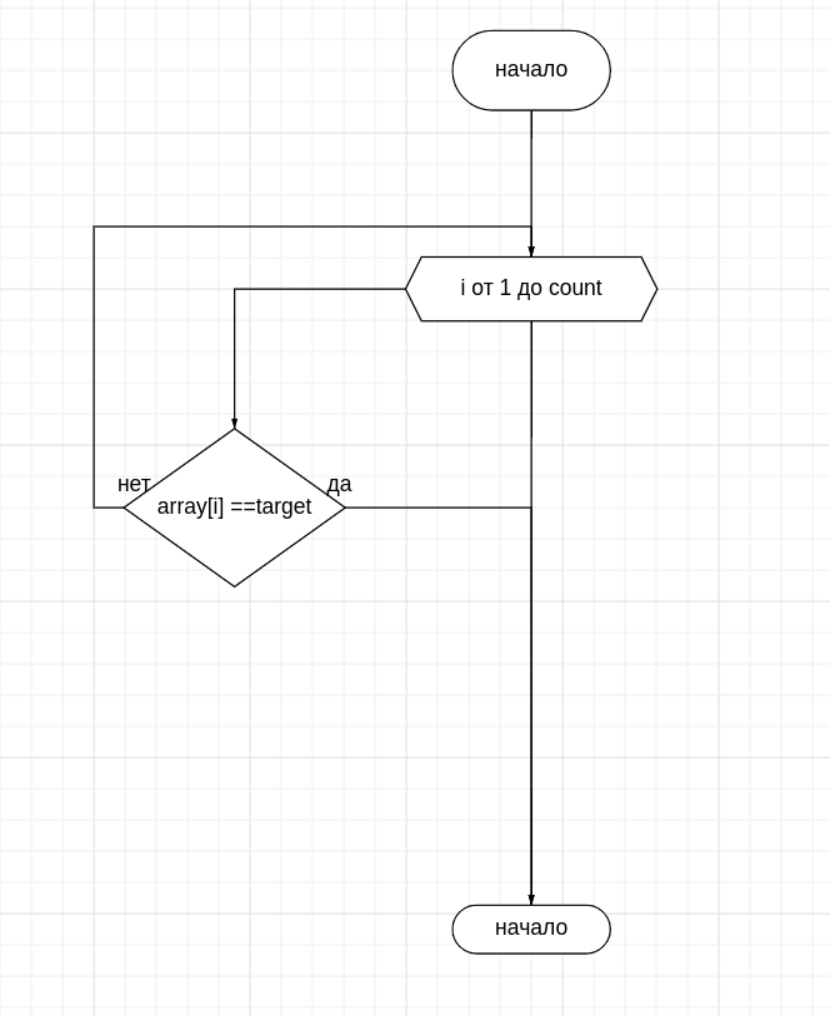

Рис. 1: Блок схема алгоритма.

4 OpenMP

$\#pragma\ omp\ parallel\ num\_threads(threads)\ shared(array,\ count)\ default(none)$

1. $omp\ parallel$ – показывает, что следующий блок кода будет выполняться с использованием нескольких потоков

2. $num\_threads(threads\_num)$ – задаёт количество потоков равное значению переменной threads

3. $shared(array,\ count)$ - указывает, что переменные array, count должны совместно использоваться для всех потоков. Если данный параметр не будет указана, то её аргументам будут присвоены правила по умолчанию (в данном случае они будут являться общими, так как все объявлены вне блока параллельных вычислений)

4. $default(none)$ - требует, чтобы каждая переменная, на которую ссылаются в блоке параллельных вычислений и которая не имеет предопределенного атрибута совместного использования данных, имела свой атрибут, явно определенный ранее.

5. $omp\ for$ – показывает, что следующий цикл for будет выполняться параллельно, несколькими потоками сразу.

5 Анализ эффективности и ускорения

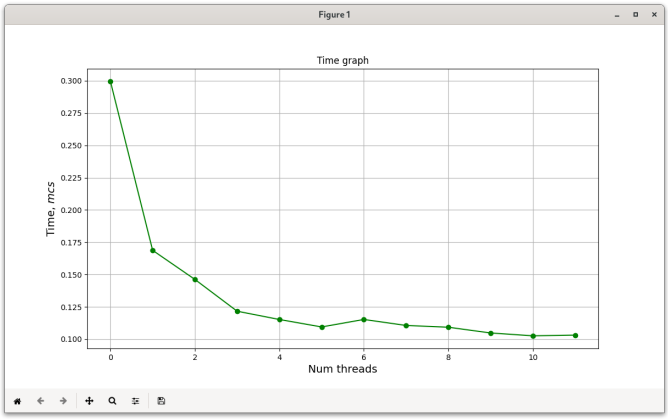

Рис. 2: График зависимости времени выполнения алгоритма от колличества потоков.

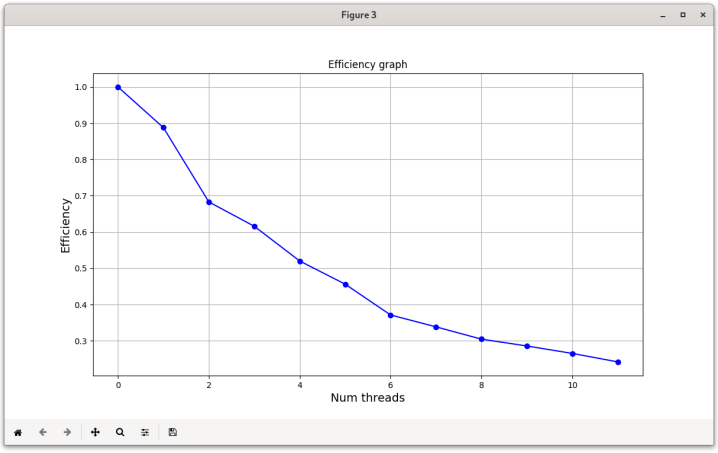

Рис. 3: График эффективности алгоритма от колличества потоков.

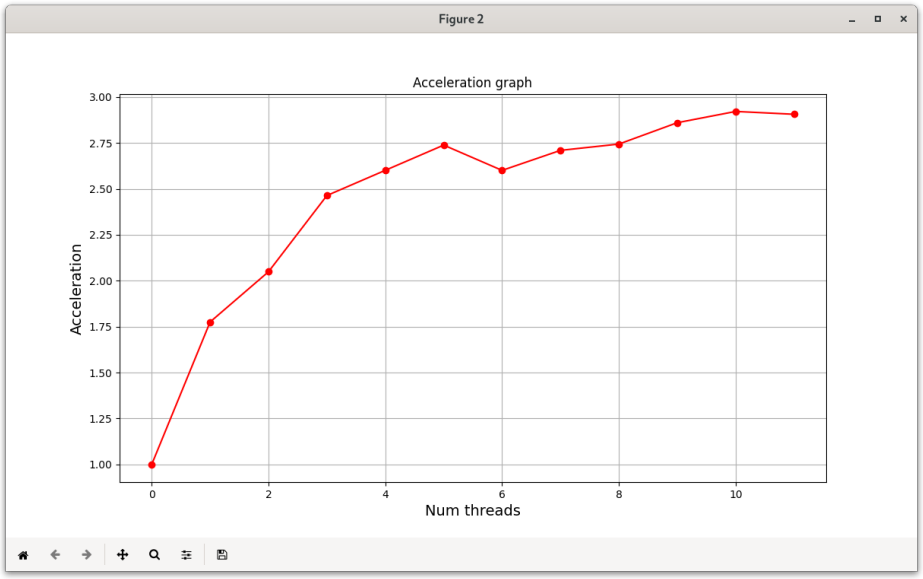

Рис. 4: График зависимости ускорения от колличества потоков.

6 Код программы

```
int main(int argc, char** argv)
{
    int n = omp_get_num_procs();

    printf("%d \n", n);

    int count = 200000; ///< Number of array elements
    const int threads = 12; ///< Number of parallel threads to use
    const int random_seed = 64; ///< RNG seed
    const int rep = 10000;
    int target = 0;
    int index = -1;
    const int u = 5;
    int** array = 0; ///< The array we need to find the max in
    int max = -1; ///< The maximal element
    srand(time(NULL));

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

    std::ofstream out("./points.txt");
    if (!out.is_open())
        return 1;

    array = (int**)malloc(rep*sizeof(int*));

    for (int i = 0; i < rep; i++) {
        array[i] = (int *) malloc((count+1) * sizeof(int));
        for (int j = 0; j < count; j++) {
            array[i][j] = rand();
        }
        array[i][count] = rand() % count;
    }
```

7 Заключение

В рамках данной работы был изучен приведённый алгоритм, оценена его вычислительная сложность в однопоточном варианте, а также бы-

```
    double start, finish;
    int *array2;

    for(int r = 1; r <= threads; r++) {
        start = omp_get_wtime();
        for (int t = 0; t < rep*u; t++) {
            count = 20000;
            array2 = array[t/u];
            target = array2[count];

            //printf("%d \n", target);

            #pragma omp parallel num_threads(r) shared(array2, count, target, index) reduction(max: max) default(none)
            {
                #pragma omp for
                for (int i = 0; i < count; i++) {
                    if (array2[i] == target) {
                        index = i;
                        count = 0;
                    };
                }
            }

        //printf("%d %d \n", target, array2[index]);
        }

        finish = omp_get_wtime();
        printf("%d %lf\n", r, (finish - start) * 1000000 / rep);

        writeFile(r, (finish - start) * 1000000 / rep, out);
    }

    out.close();

    return 0;
}
```

Рис. 5: Код программы.

ла построена программа, выполняющая параллельные вычисления. Для успешного выполнения лабораторной работы был изучен и использован ряд директив OpenMP. По результатам работы программы были построены графики зависимости времени работы, ускорения и эффективности от числа потоков. По данным графикам видно, что параллельные вычисления дают большой выигрыш по времени в работе программы.