Национальный исследовательский ядерный университет «МИФИ» (Московский Инженерно–Физический Институт)

Кафедра №42 «Криптология и кибербезопасность»

**Лабораторная работа №3:**

**«Реализация алгоритма с использованием технологии OpenMP»**

Антон Гатченко Б22-525

2024 г.

*Используемая рабочая среда:*

* Процессор - AMD Ryzen 5 5600H (laptop), 6c/12t
* Оперативная память – DDR4 16 ГБ
* ОС - Windows 10 Pro 22H2 19045.4780, 64 bit
* IDE - CLion 2024.2.1, G++ 13.1, OpenMP 201511

*Анализ последовательного алгоритма*:

Значения некоторого массива, стоящие один от другого на некотором расстоянии , сравниваются и сортируются между собой, после этого процедура повторяется с уменьшением , пока .

Массив размером элементов заполнялся числами от INT\_MIN до INT\_MAX.

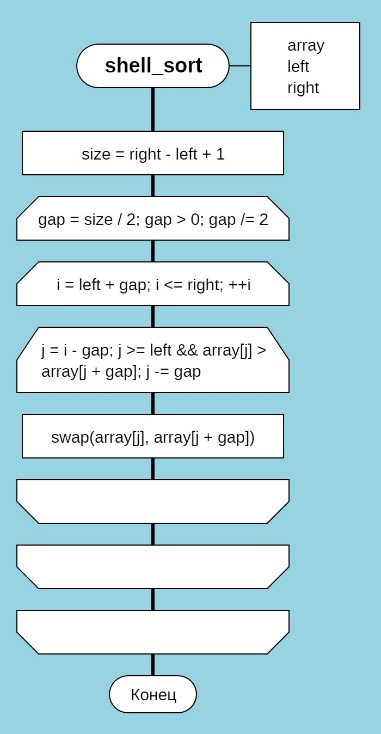
Теоретическая временная сложность: в данной реализации; разной подборке значений d соответствует разная временная сложность. Например, для последовательности Пратта сложность алгоритма составляет .

Теоретическое ускорение параллельных вычислений:

Объединение отсортированных блоков с помощью функции слияния дополнительно займет времени, причём ускорение этого процесса будет с всё уменьшающимся количеством блоков, и, как следствие, потоков.

Теоретическая эффективность параллельных вычислений:

Блок-схема алгоритма:



*Анализ параллельного алгоритма и директивы OpenMP в функциях:*

Так как в сортировке Шелла следующий результат прохода зависит от предыдущего, был использован tile-based принцип для распараллеливания. Исходный массив был поделен на блоки, которые сортировались с помощью сортировки Шелла. Затем блоки объединялись с помощью функции слияния.

1.

void tile\_based\_shell\_sort(vector<int> &array, const int num\_blocks){  
 const int size = array.size();  
 const int block\_size = size / num\_blocks;  
#pragma omp parallel for num\_threads(num\_blocks) shared(num\_blocks, block\_size, array, size) default(none)  
 for (int i = 0; i < num\_blocks; ++i){  
 const int start = i \* block\_size;  
 const int end = (i == num\_blocks - 1) ? size - 1 : start + block\_size - 1;  
 shell\_sort(array, start, end);  
 }  
 parallel\_merge\_blocks(array, size, num\_blocks);  
}

В функции tile\_based\_shell\_sort массив делится на равномерные блоки, на которых выполняется сортировка Шелла. После этого происходит сортировка этих блоков с помощью функции parallel\_merge\_blocks.

2.

void parallel\_merge\_blocks(vector<int> &array, const int size, int num\_blocks){  
 int block\_size = size / num\_blocks;  
  
 while (block\_size < size){  
#pragma omp parallel for num\_threads(num\_blocks) shared(block\_size, size, num\_blocks, array) default(none)  
 for (int i = 0; i < num\_blocks; i += 2){  
 const int left = i \* block\_size;  
 const int mid = left + block\_size - 1;  
 const int right = min(left + 2 \* block\_size - 1, size - 1);  
  
 if (mid < right){  
 merge(array, left, mid, right);  
 }  
 }  
 block\_size \*= 2;  
 num\_blocks /= 2;  
 }  
}

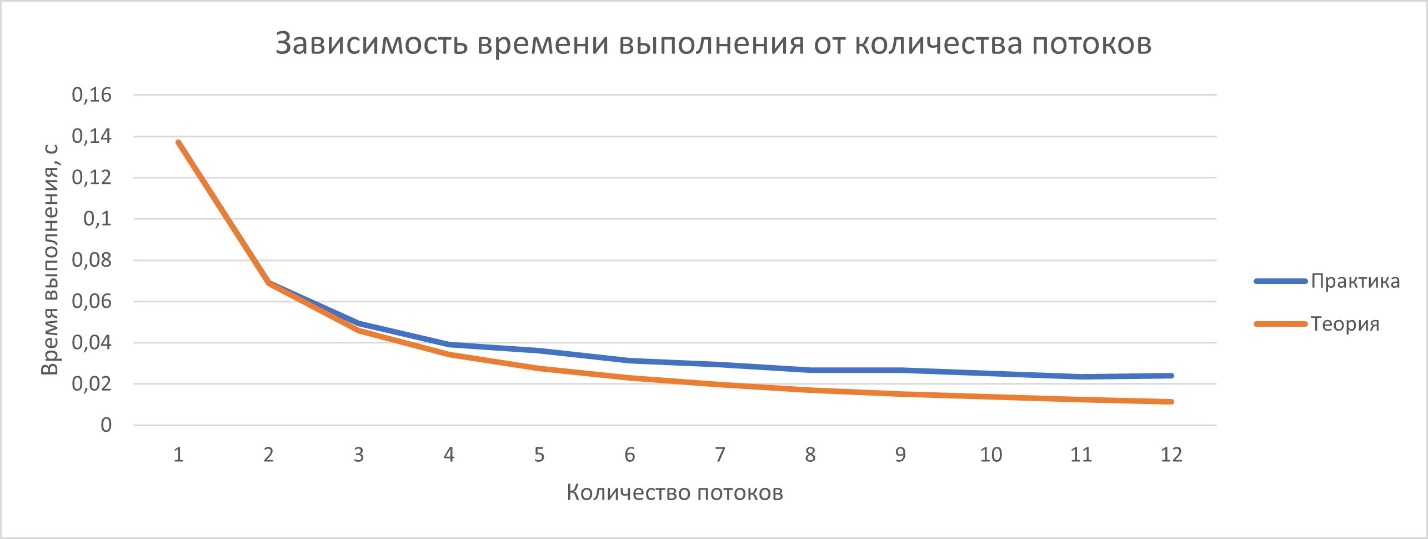
В функции parallel\_merge\_blocks объединяются отсортированные «блоки» внутри массива с помощью merge. После одной итерации количество блоков уменьшается вдвое, а размер одного блока возрастает в 2 раза. Количество потоков также зависит от количества блоков, т.к. последующие итерации зависят от предыдущих.

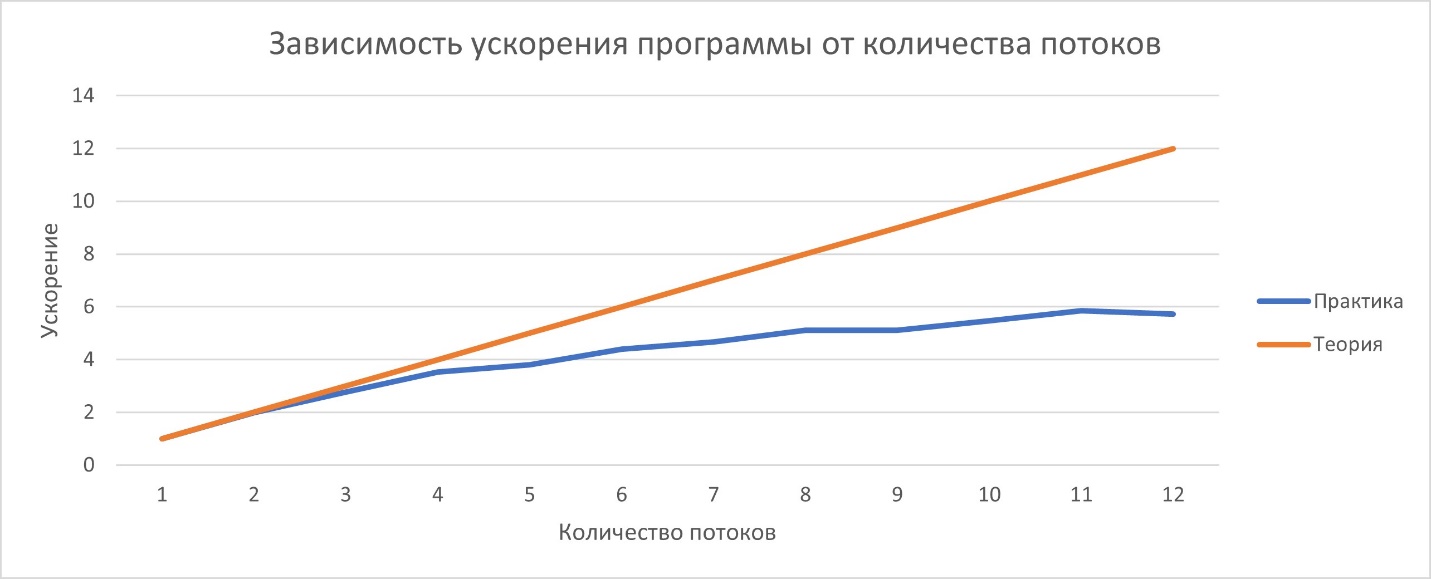
Директива parallel – создание потоков/нитей. Блок (в фигурных скобках) после данной директивы будет распараллелен.

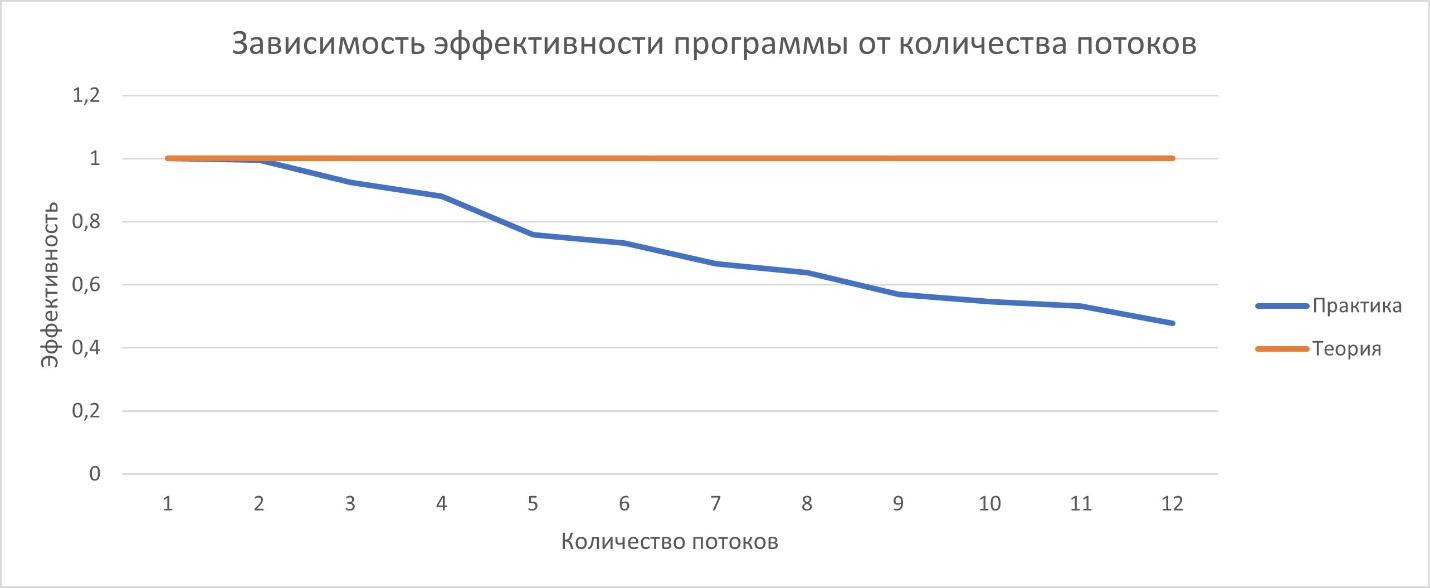
Директива for - явное распараллеливание цикла for, находящегося сразу после директивы, при этом каждый поток начнет со своего индекса. Без этой директивы каждый поток будет идти от начала и до конца цикла, проходя все индексы.

Директива parallel for – сокращение для одновременного использования двух вышеописанных директив.

*Графики выполнения программы при различном количестве потоков:*







|  |  |
| --- | --- |
| Время выполнения алгоритмов при 1 потоке | |
| **Последовательный** | **Параллельный** |
| 0,1371 | 0,1378 |

*Заключение:*

В ходе данной лабораторной работы был изучен процесс использования технологии OpenMP для организации параллельных вычислений с нуля на примере функции сортировки Шелла. Программа была запущена при разном количестве потоков, что позволило оценить влияние распараллеливания на производительность и время выполнения алгоритма.

Результаты тестов показали, что теоретические предположения об ускорении и эффективности выполнения программы при увеличении числа потоков расходятся с практическими (особенно при количестве потоков).

Ускорение параллелизации начинает постепенно снижаться при количестве потоков больше 4, что может объясняться тем, что потоки выполняют свою задачу не в одно и то же время (разные части массива могут быть более затратными), сторонними процессами системы, влиянием планировщика ОС на выделяемые программе ресурсы, а также использование логических ядер вместе с физическими.

*Приложение:*

1. Исходный код программы с измерением времени работы программы:

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <random>  
#include <cassert>  
#include <iomanip>  
#include <fcntl.h>  
#include <omp.h>  
#include <set>  
  
#define SEED 920215  
#define SEED\_INC 12345  
#define RUNS\_NUM 20  
  
using std::cout, std::wcout, std::cin, std::endl, std::vector, std::set, std::swap, std::min;  
  
void generate\_random\_array(vector<int> &array, const int seed){  
 std::mt19937\_64 gen(seed);  
 std::uniform\_int\_distribution dist(INT\_MIN, INT\_MAX);  
  
 for (int &el: array){  
 el = dist(gen);  
 }  
}  
  
void shell\_sort(vector<int> &array, const int left, const int right){  
 const int size = right - left + 1;  
  
 for (int gap = size / 2; gap > 0; gap /= 2){  
 for (int i = left + gap; i <= right; ++i){  
 for (int j = i - gap; j >= left && array[j] > array[j + gap]; j -= gap){  
 swap(array[j], array[j + gap]);  
 }  
 }  
 }  
}  
  
void merge(vector<int> &array, const int left, const int mid, const int right){  
 vector<int> temp(right - left + 1);  
 int i = left, j = mid + 1, k = 0;  
  
 while (i <= mid && j <= right){  
 if (array[i] <= array[j]){  
 temp[k++] = array[i++];  
 } else{  
 temp[k++] = array[j++];  
 }  
 }  
 while (i <= mid){  
 temp[k++] = array[i++];  
 }  
 while (j <= right){  
 temp[k++] = array[j++];  
 }  
 for (i = left; i <= right; ++i){  
 array[i] = temp[i - left];  
 }  
}  
  
void parallel\_merge\_blocks(vector<int> &array, const int size, int num\_blocks){  
 int block\_size = size / num\_blocks;  
  
 while (block\_size < size){  
#pragma omp parallel for num\_threads(num\_blocks) shared(block\_size, size, num\_blocks, array) default(none)  
 for (int i = 0; i < num\_blocks; i += 2){  
 const int left = i \* block\_size;  
 const int mid = left + block\_size - 1;  
 const int right = min(left + 2 \* block\_size - 1, size - 1);  
  
 if (mid < right){  
 merge(array, left, mid, right);  
 }  
 }  
 block\_size \*= 2;  
 num\_blocks /= 2;  
 }  
}  
  
void tile\_based\_shell\_sort(vector<int> &array, const int num\_blocks){  
 const int size = array.size();  
 const int block\_size = size / num\_blocks;  
#pragma omp parallel for num\_threads(num\_blocks) shared(num\_blocks, block\_size, array, size) default(none)  
 for (int i = 0; i < num\_blocks; ++i){  
 const int start = i \* block\_size;  
 const int end = (i == num\_blocks - 1) ? size - 1 : start + block\_size - 1;  
 shell\_sort(array, start, end);  
 }  
 parallel\_merge\_blocks(array, size, num\_blocks);  
}  
  
void time\_algorithm(vector<int> &array, const int &threads\_num){  
 for (int t = 1; t <= threads\_num; t++){  
 double time\_spent = 0;  
 int seed = SEED;  
 for (int i = 0; i < RUNS\_NUM; i++){  
 generate\_random\_array(array, seed);  
 seed += SEED\_INC;  
  
 const double start = omp\_get\_wtime();  
 tile\_based\_shell\_sort(array, t);  
 const double end = omp\_get\_wtime();  
  
 time\_spent += end - start;  
 }  
 wcout << time\_spent / RUNS\_NUM << endl;  
 }  
}  
  
void time\_consecutive\_algorithm(vector<int> &array){  
 double time\_spent = 0;  
 int seed = SEED;  
  
 for (int i = 0; i < RUNS\_NUM; i++){  
 generate\_random\_array(array, seed);  
 seed += SEED\_INC;  
  
 const double start = omp\_get\_wtime();  
 shell\_sort(array, 0, array.size());  
 const double end = omp\_get\_wtime();  
  
 time\_spent += end - start;  
 }  
 wcout << time\_spent / RUNS\_NUM << endl;  
}  
  
int main(){  
 const int size = static\_cast<int>(pow(10, 6));  
 vector<int> array(size);  
 const int threads\_num = omp\_get\_num\_procs();  
  
 assert(size > threads\_num);  
 \_setmode(\_fileno(stdout), \_O\_U16TEXT);  
 wcout << "OpenMP: " << \_OPENMP << endl;  
  
 wcout << L"Последовательный shell sort:" << endl;  
 time\_consecutive\_algorithm(array);  
  
 wcout << L"Параллельный shell sort:" << endl;  
 time\_algorithm(array, threads\_num);  
  
 return 0;  
}

1. Таблица времени работы программы

|  |  |  |
| --- | --- | --- |
| Время работы программы | | |
| **Количество потоков** | **Практика** | **Теория** |
| 1 | 0,1372 | 0,1372 |
| 2 | 0,0689 | 0,0686 |
| 3 | 0,04945 | 0,045733333 |
| 4 | 0,039 | 0,0343 |
| 5 | 0,03615 | 0,02744 |
| 6 | 0,03125 | 0,022866667 |
| 7 | 0,0294 | 0,0196 |
| 8 | 0,02685 | 0,01715 |
| 9 | 0,0268 | 0,015244444 |
| 10 | 0,0251 | 0,01372 |
| 11 | 0,02345 | 0,012472727 |
| 12 | 0,02395 | 0,011433333 |

1. Таблица ускорения программы

|  |  |  |
| --- | --- | --- |
| Ускорение программы | | |
| **Количество потоков** | **Практика** | **Теория** |
| 1 | 1 | 1 |
| 2 | 1,991291727 | 2 |
| 3 | 2,774519717 | 3 |
| 4 | 3,517948718 | 4 |
| 5 | 3,795297372 | 5 |
| 6 | 4,3904 | 6 |
| 7 | 4,666666667 | 7 |
| 8 | 5,109869646 | 8 |
| 9 | 5,119402985 | 9 |
| 10 | 5,466135458 | 10 |
| 11 | 5,850746269 | 11 |
| 12 | 5,728601253 | 12 |

1. Таблица эффективности программы

|  |  |  |
| --- | --- | --- |
| Эффективность программы | | |
| **Количество потоков** | **Практика** | **Теория** |
| 1 | 1 | 1 |
| 2 | 0,995645864 | 1 |
| 3 | 0,924839906 | 1 |
| 4 | 0,879487179 | 1 |
| 5 | 0,759059474 | 1 |
| 6 | 0,731733333 | 1 |
| 7 | 0,666666667 | 1 |
| 8 | 0,638733706 | 1 |
| 9 | 0,568822554 | 1 |
| 10 | 0,546613546 | 1 |
| 11 | 0,531886024 | 1 |
| 12 | 0,477383438 | 1 |