НИЯУ МИФИ. Лабораторная работа №3. Никифоров Степан, Б21-502. 2023.

Среда разработки

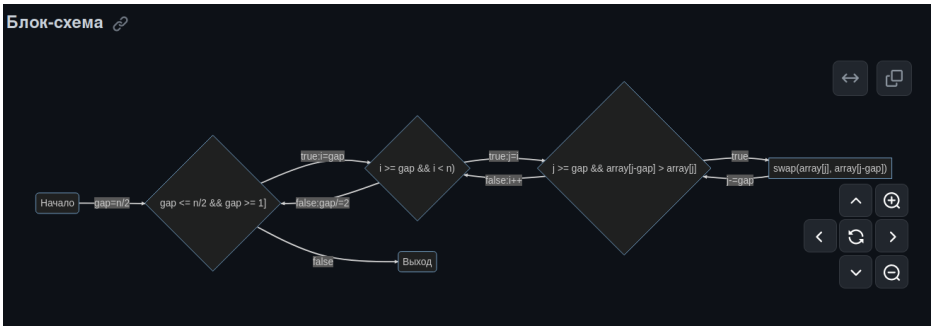
Временная оценка алгоритма

- Лучший случай O(n*log(n)) - если массив уже отсортирован
- Худший случай O(n^2) - когда последний вложенный цикл выполняется полностью(что никогда не происходит)

Анализ алгоритма

Принцип работы

Блок схема



BlockScheme

Значение директив

```
 #pragma omp parallel for shared(gap, count, array) private(i, j, tmp, part) default(none) num_threads(threads)
```

Задается обасть параллельного цикла, с количеством тредов `threads`. Переменные `array`, `count` и `gap` объявляются общими для всех тредов и непараллельной части алгоритма. Все новые переменные без явного указания класса не разрешены. Переменные `i`, `j`, `tmp` и `part` объявляется индивидуальной для каждого треда. Область - `цикл for`

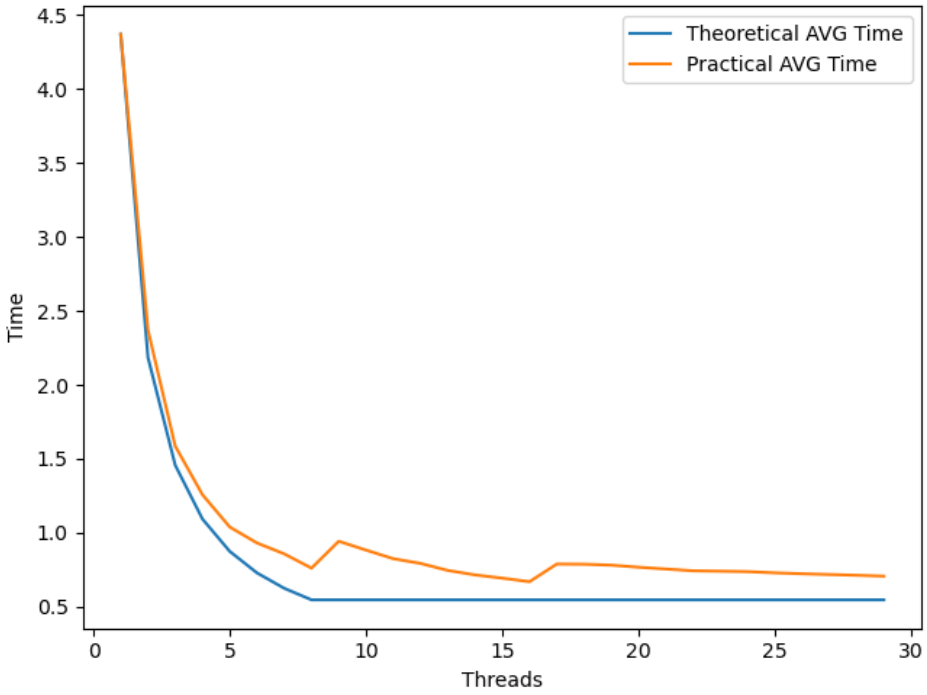
Эта директива необходима для распараллеливания сортировки элементов массива, которые отстоят друг от друга на расстоянии `gap`, потому что они не пересекаются с остальными и соответственно уменьшения время всей сортировки.

Параллельный алгоритм

200 раз генерируется случайный массив из ста тысяч элементов с разным сидом, чтобы усреднить худшие и лучшие случаи.
Всего эксперимент занял два часа

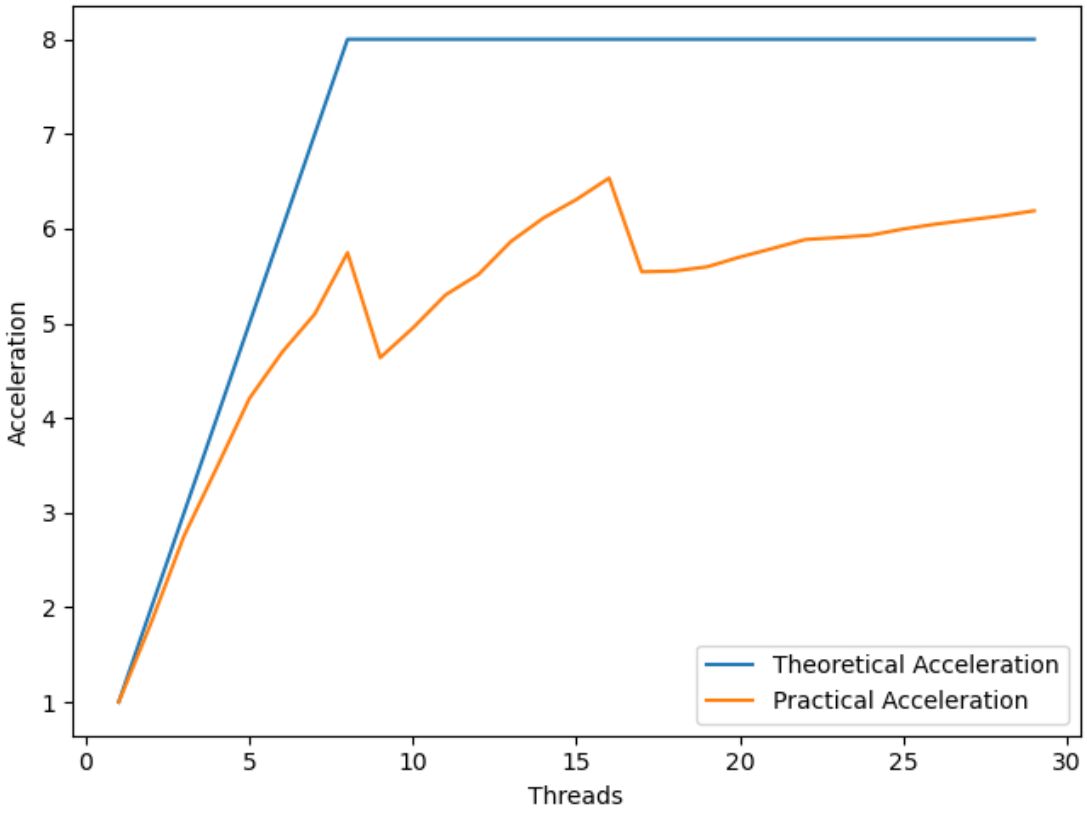
```
~/studying/parprog/lab3 | main ?1                    >:3 1h 42m 49s | 13:46:56
❯ ls
AVG_time.png  a.out*            efficiency.png  logger.py  parallel_core.py
__pycache__/  acceleration.png  lab3.c          logs/
```

Среднее время

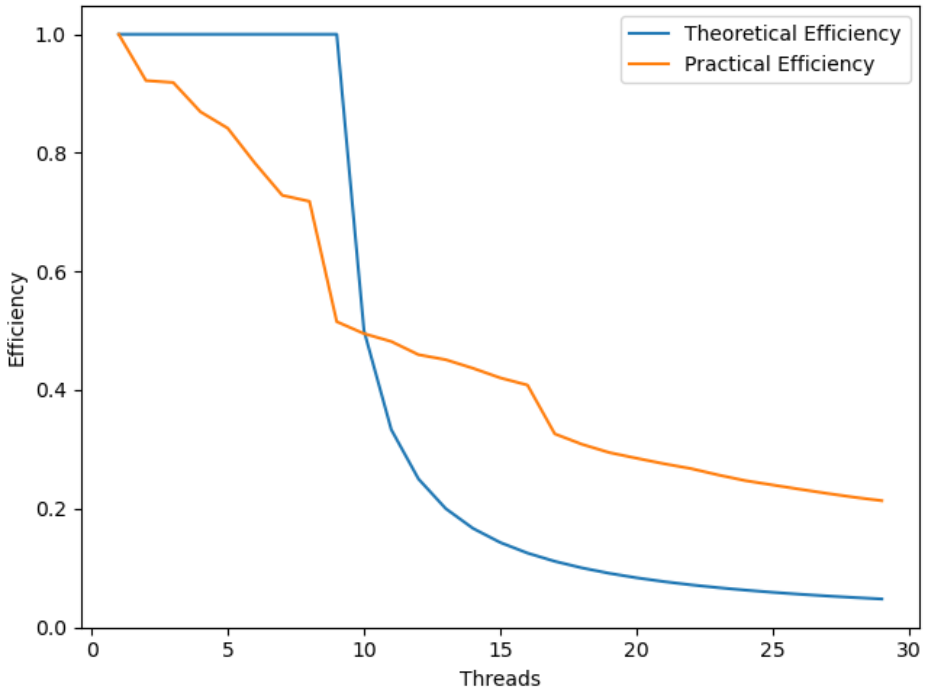


AvgTime

Среднее ускорение



Средняя эффективность



Efficiency

Заключение

В данной работе я разработал и реализовал параллельный алгоритм сортировки Шелла. Ускорение возможно, потому что во втором вложенном цикле происходят сортировки пузырьком для элементов отстающих друг от друга на фиксированную величину `gap`, а следовательно все элементы, стоящие на индексах `0..gap-1`, образуют непересекающиеся множества.

Анализ графиков показал, что: - Как и ожидалось, после 16 потоков ускорения не происходит. -

Приложение

Оценка работы последовательной программы производилось при использовании параллельной программы с одним потоком.

Оценка работы параллельной программы

```
        #include <omp.h>
        #include <stdio.h>

        double shellsort(int* array, int count, int threads){
            double t1, t2;
            int i, j, tmp, part;
            t1 = omp_get_wtime();
            for(int gap = count/2; gap > 0; gap /= 2){
                #pragma omp parallel for shared(gap, count, array)
private(i, j, tmp, part) default(none) num_threads(threads)
                for(i = 0; i < gap; i++){
                    for(part = i + gap; part < count; part += gap){
```

```
                        for(j=part; j>i  && array[j-gap] > array[j]; j-
=gap){
                            tmp = array[j];
                            array[j] = array[j-gap];
                            array[j-gap] = tmp;
                        }
                    }
                }
            }
            t2 = omp_get_wtime();
            return t2 - t1;
        }

        int main(int argc, char** argv)
        {
          const int count = 10000000;      ///< Number of array elements
          const int target = 16;           ///< Number to look for

          int* array = 0;                  ///< The array we need to find
the max in
          int  index =
-1;                ///< The index of the element we need
          if (argc < 3){
            puts("USAGE ./a.out {THREADS_NUM} {SEED}");
            return -1;
          }
          const int threads = atoi(argv[1]);          ///< Number of
parallel threads to use
          const int random_seed = atoi(argv[2]); ///< RNG seed

          /* Initialize the RNG */
          srand(random_seed);

          /* Generate the random array */

          /*
           * We can multithread array filling
           * */

          array = (int*)malloc(count*sizeof(int));
          for(int i=0; i<count; i++){
            array[i] = rand();
          }
          double t = shellsort(array, count, threads);
          printf("%g", t);

          return 0;
        }
```