МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

«Национальный исследовательский ядерный университет «МИФИ»»

ЛАБОРАТОРНАЯ РАБОТА №3:

«Реализация алгоритма с использованием технологии OpenMP»

Аверин Владислав

Группа Б19-505

Октябрь, 2021

Содержание

Характеристики лабораторного оборудования

Процессор: 11th Gen Intel Core i7-1185G7 3.00Ghz (8 CPUs)

RAM: 16Гб DDR4 3200МГц

Используемая версия _OpenMP: 201511 (December, 2015)

Операционная система: OS Linux Manjaro KDE Plasma 5.22.5; версия ядра: 5.10.68-1-MANJARO (64-бита), работа от сети

Редактор кода: Visual Studio Code 1.60.1

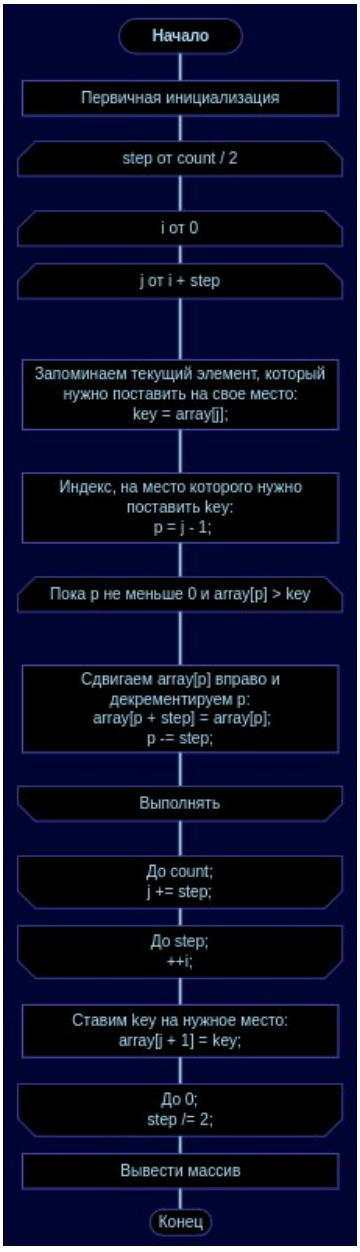
Компилятор: GCC 11.1.0

Параметры командной строки: -O0 -fopenmp (без оптимизации и отладочной информации для чистоты эксперимента)

Реализация последовательного алгоритма

В данной лабораторной работе было предложено распараллелить алгоритм Шелла - алгоритм сортировки, основанный на сортировке вставками. Идея сортировки в том, чтобы не сортировать сразу элементы, стоящие рядом друг с другом (что делает Insertion Sort), но сделать это несколько раз с определенным уменьшающимся шагом расстояния между сравниваемыми элементами. То есть по сути, на каждом шаге мы делим последовательность на step подпоследовательностей, элементы которых стоят на расстоянии step друг от друга, и производим сортировку вставками в этих локальных подпоследовательностях. Математически доказано, что такой способ последовательного применения Insertion Sort уменьшает кол-во перестановок и инверсий в исходном массиве, так что при последнем применении сортировки вставками для шага 1 кол-во элементов, стоящих не на своих местах, будет минимально.

Блок-схема и код последовательного алгоритма:




```
for (int step = _SIZE / 2; step > 0; step /= 2)
{
    for (int i = 0; i < step; ++i)
    {
        for (int j = i + step; j < _SIZE; j += step)
        {
            int key = object[j], p = j - step;
            {
                while (p >= 0 && object[p] > key)
                {
                    object[p + step] = object[p];
                    p -= step;
                }
                object[p + step] = key;
            }
        }
    }
```

Временная сложность последовательного алгоритма

- Сложность алгоритма в худшем случае: $O(N^2)$;
- Сложность алгоритма в лучшем случае: $O(N * log^2 N)$;
- Сложность алгоритма в среднем случае напрямую зависит от выбранных шагов; в нашем случае была выбрана последовательность, предложенная самим Шеллом: step = count / 2^p, где p - текущий шаг; в таком случае, сложность в среднем случае (без учета свойств массива, таких как частичная упорядоченность, или частота появления одинаковых элементов) оценивается как $O(N^2)$

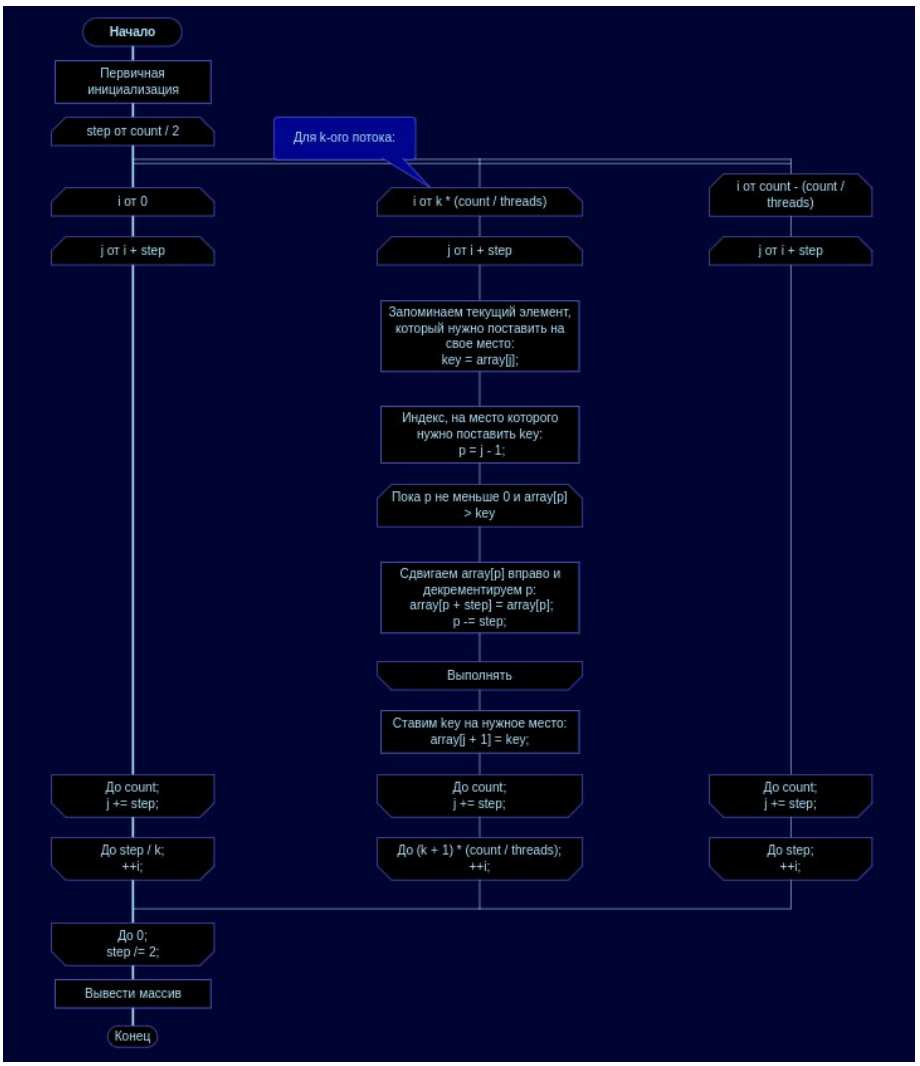
Примечание: существуют и другие способы выбора шага. Например, для предложенной Холлом Хиббардом последовательности {i}, члены которой удовлетворяют неравенству: $2^i - 1 <= N$ средняя сложность будет $O(N^{3/2})$

Реализация параллельного алгоритма

Сортировка Шелла очень хорошо распараллеливается за счет того, что все подпоследовательности, образовывающиеся на каждом шаге, полностью не зависимы друг от друга. Поэтому нам достаточно распределить все эти подпоследовательности по процессорам, чтобы каждый из них производил сортировку в своей группе.

Из собственных наблюдений был сделан вывод о том, что поднастройка потоков в дерективе #pragma omp parallel через свойство num_threads() по всей видимости генерирует потоки заново при каждой новой итерации шага, т.к. ускорения больше 2 получить не удалось. Но при *предварительной* инициализации кол-ва потоков через функцию omp_set_num_threads() производительность возросла, из-за того, что параллельный участок кода генерирует потоки только один раз и каждый раз только перераспределяет индексы распараллеливания, не тратя время на новую генерацию и удаление старых потоков.

Блок-схема и код параллельной реализации:




```
    omp_set_num_threads(_THREADS);
#pragma omp parallel shared(object)
    {
        for (int step = _SIZE / 2; step > 0; step /= 2)
        {
#pragma omp for
            for (int i = 0; i < step; ++i)
            {
                //printf("step: %d; thread: %d; i: %d;\n", ste
                for (int j = i + step; j < _SIZE; j += step)
                {
                    int key = object[j], p = j - step;
                    {
                        while (p >= 0 && object[p] > key)
                        {
                            object[p + step] = object[p];
                            p -= step;
                        }
                        object[p + step] = key;
                    }
                }
            }
        }
    }
```

Временная сложность параллельного алгоритма

Исходя из концепции параллельной реализации, приходим к выводу, что за время сортировки вставками одной подпоследовательности p процессоров в среднем сортируют p подпоследовательностей. Соответственно, каждый из процессоров обрабатывает N / p подпоследовательностей за всю сортировку (не будем учитывать разное время обработки подпоследовательностей и то, что с уменьшением длины шага до значения < p возможность продуктивно распараллелить итерации резко падает).

То есть сложность параллельного алгоритма в среднем случае: $\boldsymbol{O(N^2 / p)}$;

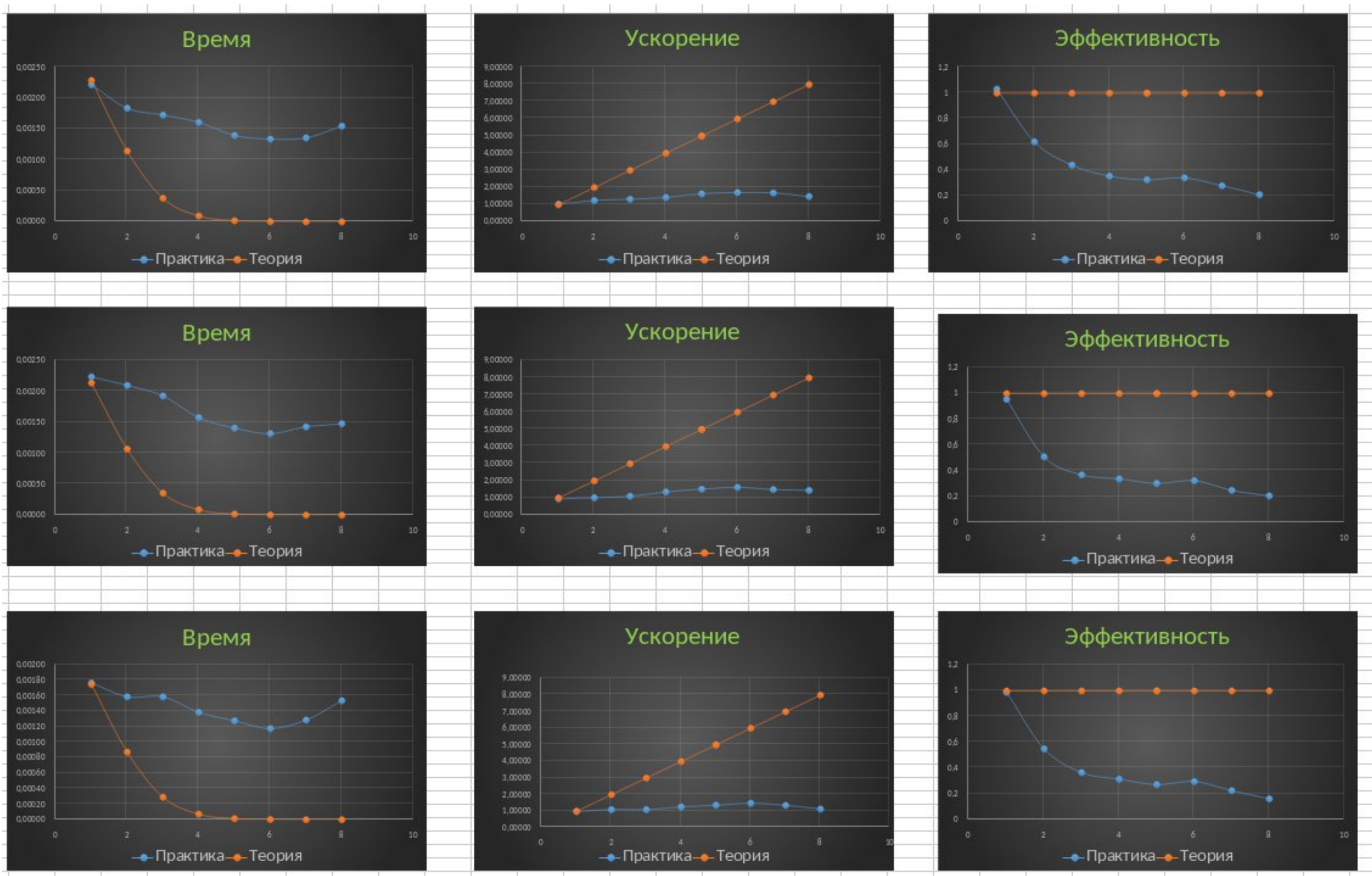
Экспериментальные данные

На практике попытаемся учесть хотя бы некоторые свойства массива: самые тривиальные и легко реализуемые: размер массива и частота встречи одинаковых элементов.

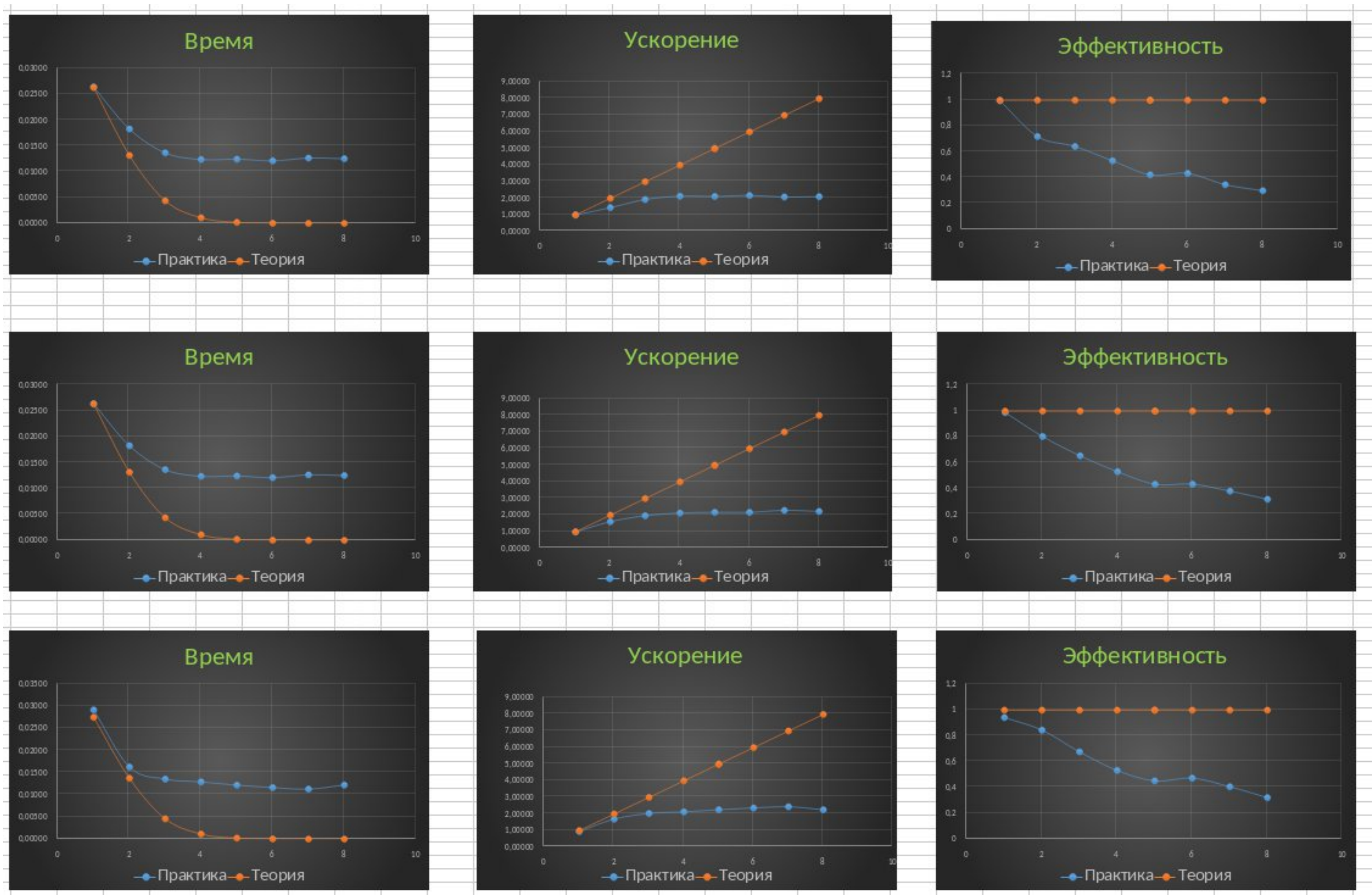
В таблице значений представлены усредненные результаты опытов на 100 различных между собой массивах (среднее из 10 опытов на каждом) на размерах 1e4, 1e5, 1e6 и 1e7, а так же различной частотой вхождения: 0.1, 1 и 10. Частота вхождения показывает, во сколько раз размер массива больше возможного диапазона значений (Т.е. чем больше частота, тем больше вероятность существования дубликатов).

Array count: 100 Seed: 666666 + 1000													
Array size		10000			100000			1000000			10000000		
Frequency		0,1	1	10	0,1	1	10	0,1	1	10	0,1	1	10
Time	Consecutive	0,00229	0,00214	0,00175	0,02636	0,02720	0,02754	0,37476	0,38372	0,37672	5,9461	6,1387	6,20056
	1	0,00222	0,00224	0,00177	0,02645	0,02751	0,02921	0,37932	0,39665	0,38130	6,129183	6,60915	6,37568
	2	0,00184	0,00210	0,00159	0,01831	0,01692	0,01630	0,18830	0,18381	0,20625	3,282202	3,35351	3,25872
	3	0,00173	0,00193	0,00159	0,01368	0,01385	0,01357	0,14749	0,15307	0,12239	2,24638	2,85631	2,21921
	4	0,00161	0,00158	0,00139	0,01240	0,01279	0,01293	0,12856	0,11903	0,11154	1,81271	2,41816	1,83058
	5	0,00140	0,00141	0,00128	0,01246	0,01252	0,01220	0,11513	0,10619	0,10008	1,637345	2,09492	1,59159
	6	0,00134	0,00132	0,00118	0,01214	0,01251	0,01166	0,10275	0,10425	0,08949	1,541003	1,92882	1,5662
	7	0,00136	0,00143	0,00129	0,01267	0,01189	0,01131	0,09481	0,09572	0,08861	1,43201	1,72754	1,40973
	8	0,00155	0,00148	0,00154	0,01255	0,01221	0,01223	0,11403	0,09938	0,09879	1,47894	1,82752	1,54475
Acceleration	1	1,03153	0,95536	0,98870	0,99660	0,98873	0,94283	0,98798	0,96740	0,98799	0,97013	0,92882	0,97253
	2	1,24457	1,01905	1,10063	1,43965	1,60757	1,68957	1,99023	2,08759	1,82652	1,81162	1,83053	1,90276
	3	1,32370	1,10881	1,10063	1,92690	1,96390	2,02948	2,54092	2,50683	3,07803	2,64697	2,14917	2,79404
	4	1,4224	1,3544	1,25899	2,12581	2,12666	2,12993	2,91506	3,22373	3,37744	3,28023	2,53858	3,38721
	5	1,63571	1,5177	1,36719	2,11557	2,17252	2,25738	3,25510	3,61352	3,76419	3,63155	2,93028	3,89583
	6	1,70896	1,62121	1,48305	2,17133	2,17426	2,36192	3,64730	3,68077	4,20963	3,85859	3,18262	3,95898
	7	1,68382	1,49650	1,35659	2,08051	2,28764	2,43501	3,95275	4,00878	4,25144	4,15228	3,55343	4,39840
	8	1,47742	1,44595	1,13636	2,10040	2,22768	2,25184	3,28650	3,86114	3,81334	4,02051	3,35903	4,01396
Efficiency	1	1,031532	0,955357	0,988701	0,996597	0,988731	0,942828	0,987978	0,967402	0,987988	0,970129	0,928818	0,972533
	2	0,622283	0,509524	0,550314	0,719825	0,803783	0,844785	0,995114	1,043795	0,913261	0,90581	0,915265	0,95138
	3	0,441233	0,369603	0,366876	0,6423	0,654633	0,676492	0,846973	0,835609	1,02601	0,882323	0,71639	0,931346
	4	0,35559	0,338608	0,314748	0,531452	0,531665	0,532483	0,728765	0,805931	0,844361	0,820057	0,634646	0,846803
	5	0,327143	0,303546	0,273438	0,423114	0,434505	0,451475	0,651021	0,722705	0,752838	0,72631	0,586056	0,779165
	5	0,341791	0,324242	0,29661	0,434267	0,434852	0,472384	0,72946	0,736153	0,841926	0,771718	0,636524	0,791797
	6	0,280637	0,249417	0,226098	0,346751	0,381273	0,405836	0,658791	0,668129	0,708573	0,692046	0,592239	0,733067
	7	0,21106	0,206564	0,162338	0,300057	0,31824	0,321691	0,469501	0,551591	0,544763	0,574359	0,479862	0,573422
	8	0,128941	0,11942	0,123588	0,124575	0,123591	0,117853	0,123497	0,120925	0,123499	0,121266	0,116102	0,121567

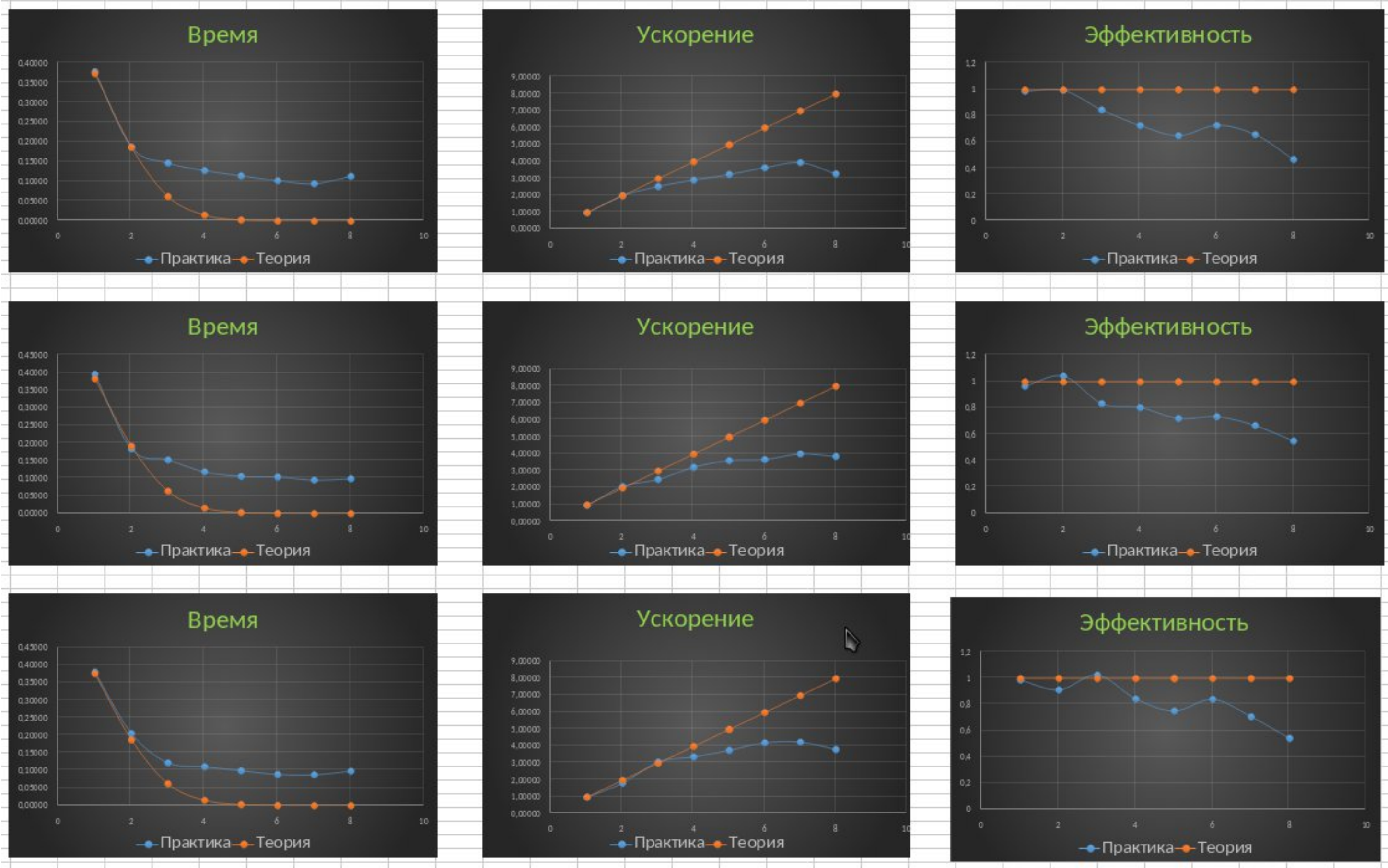
Размер: 1e4 (сверху вниз - частоты 0.1, 1 и 10)



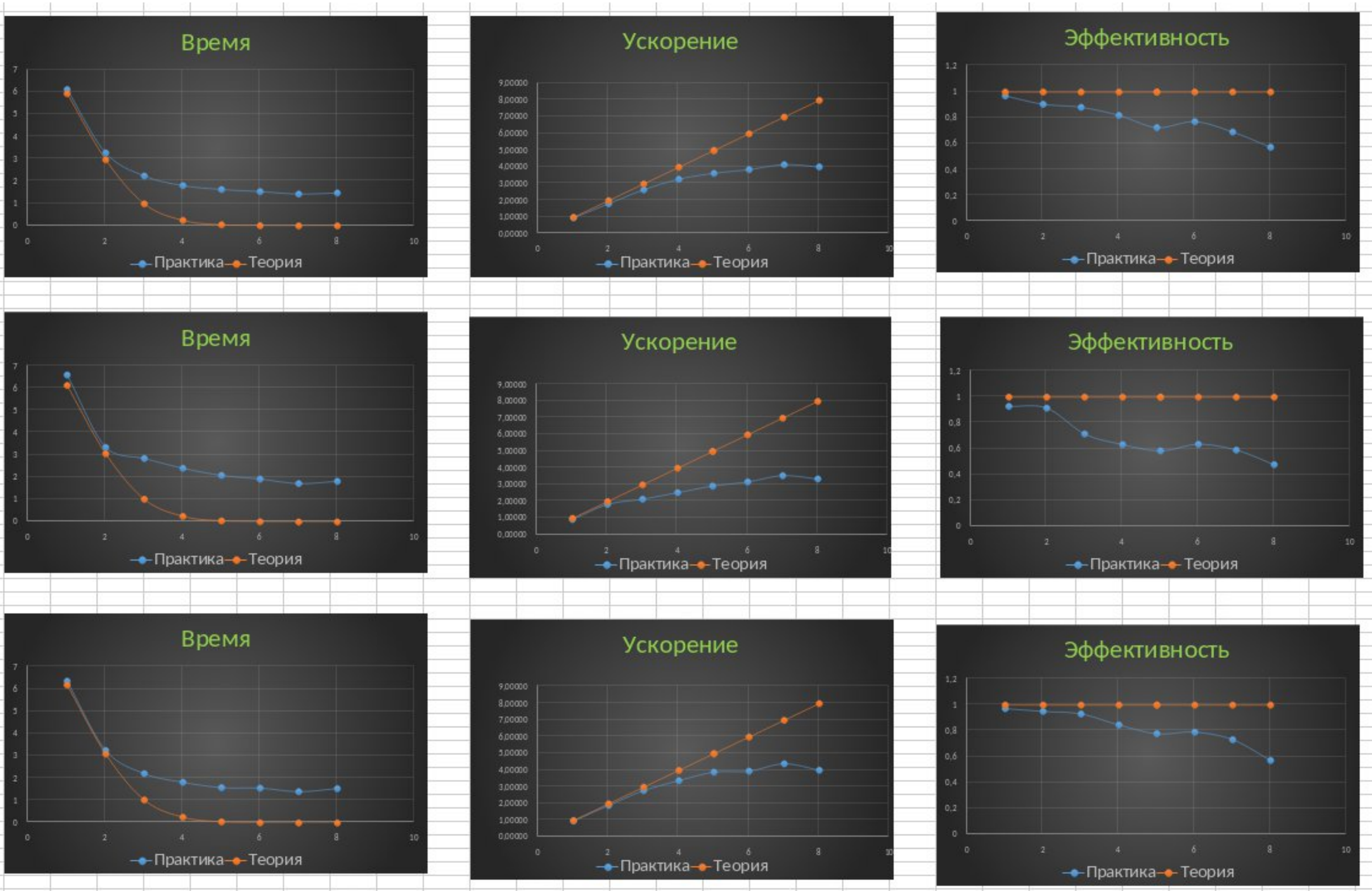
Размер: 1e5



Размер: 1e6



Размер: 1e7



Выводы

Исходя из экспериментальных и практических данных можно сделать вывод, что увеличение размера сортированного массива и частоты вхождения элементов обеспечивается бОльшая точность результатов: так, для массива размером 1e4 максимальное ускорение было ~ 1.71, а для массива из 1e7 элементов и частотой 10 практическое время очень близко к теоретическому: максимальное ускорение 4,49 а соответственно эффективность не падает отметки ниже 0.6. Можно сделать предположение, что с увеличением размера еще на порядок, результаты совпадут еще сильнее. Однако с учетом кол-ва массивов и точности измерений (усреднение каждого результата по 10 экспериментам) данная проверка очень затруднительна (занимает порядка 12 минут).