***МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ***

***Федеральное государственное автономное образовательное учреждение***

***высшего образования***

**«Национальный исследовательский ядерный университет «МИФИ»»**

*ЛАБОРАТОРНАЯ РАБОТА №1:*

*«Выделение ресурса параллелизма. Технология OpenMP»*

Аверин Владислав

Группа Б19-505

Сентябрь, 2021

***Содержание***

*Характеристики лабораторного оборудования 3*

*Реализация последовательного алгоритма 4*

*Временная сложность последовательного алгоритма 5*

*Реализация параллельного алгоритма 8*

*Временная сложность параллельного алгоритма 9*

*Экспериментальные данные 9*

*Оптимизация алгоритма 16*

*Выводы 18*

***Характеристики лабораторного оборудования***

*Пооцессор*: 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

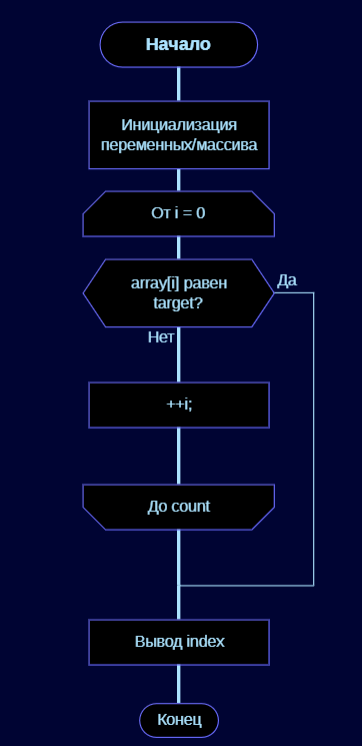
*Редактор кода*: Visual Studio Code 1.60.1

*Компилятор*: GCC 11.1.0

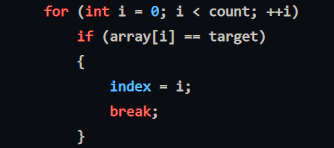
*Параметры командной строки*: -O2 -fopenmp

***Реализация последовательного алгоритма***

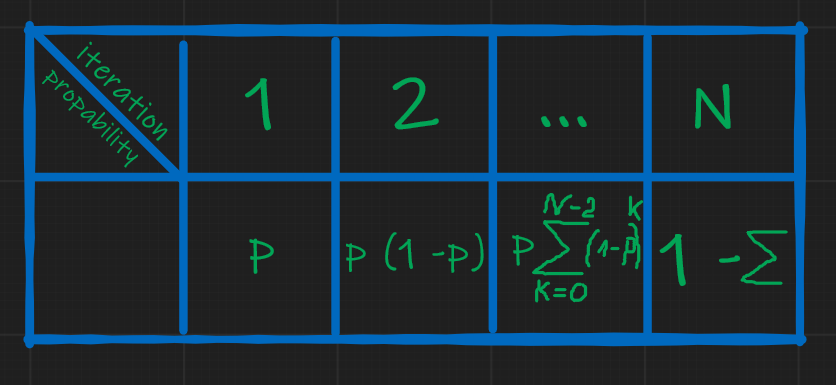
*Блок-схема последовательного алгоритма:*

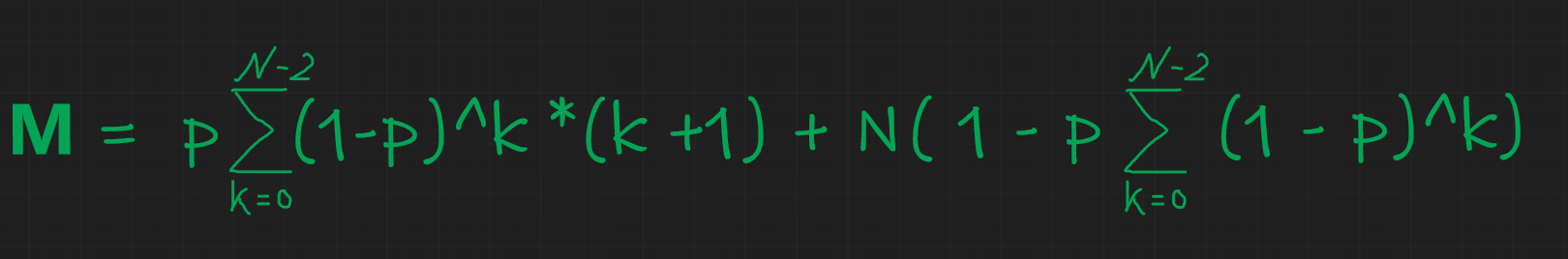


*Код последовательной реализации:*

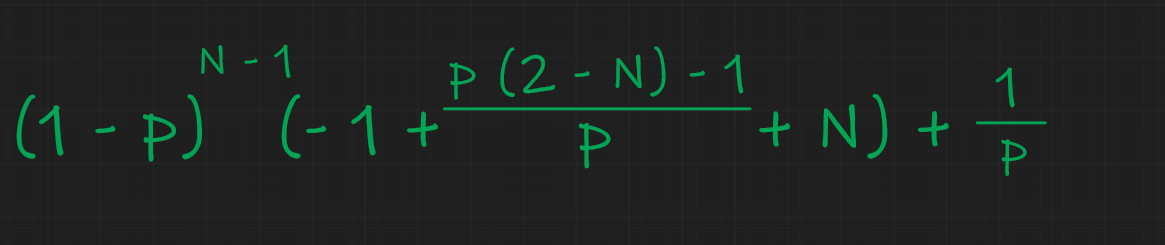


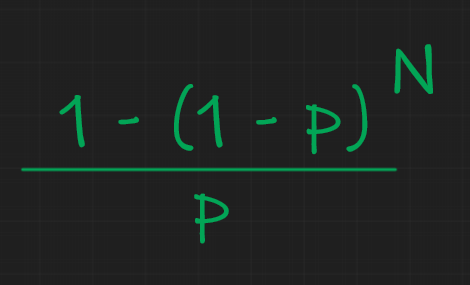
***Временная сложность последовательного алгоритма***

*Временная сложность последовательного алгоритма* зависит напрямую от того, как близко находится элемент к границе массива (левой, если индексация в цикле поиска с 0, или правой, если с count – 1). Таким образом, в лучшем случае поиск будет производиться за O(1) (только одно сравнение), а в худшем за O(N) (элемента в списке нет). Однако это все граничные случаи, которые очень тесно связаны со структурой самого массива: каков его размер, какая концентрация искомых элементов. Если принять за данность то, что в массиве размера N всего могут присутствовать N различных значений и их появление равновероятно (т.е. для каждой позиции массива вероятность ***p*** = 1/N), то в среднем потребуется N/2 операций для того, чтобы найти (или не найти) искомый элемент (т.к. в каких-то случаях он может в принципе не появиться в массиве, а когда-то быть в самом начале). Но это достаточно грубая оценка, которая, к тому же, никак не подтверждается, кроме логики. Увеличивая вероятность ***p***, например, в два раза, мы уже не сможем сказать, сколько в среднем нам необходимо будет перебрать элементов при кол-ве опытов, стремящихся к нулю. Тогда пойдем другим путем и будем исходить чисто из теории вероятности. Предположим, что вероятность появления таргета (нашего искомого элемента) в какой-либо ячейке везде одинакова и равна 0 < p < 1 (если p = 0, то поиск вообще никогда не увенчается успехом, а при p = 1 это просто O(1)). Это достаточно, как я считаю, недалекое от реальности упущение, так как вероятность в реальных условиях не зависит от других внешних параметров (например, от индекса ячейки массива), а введя неизвестную p, мы, на самом деле, можем все максимально костыльно в лоб все посчитать. Будем искать вероятность того, что элемент найдется в первой же ячейке за одну итерацию (это просто p), потом вероятность того, что гарантированно не находится в первой и находится во второй, т.е. за две итерации (p (p -1)), потом за три итерации, и так далее:

Соответственно в общем виде эту сумму можно представить как ряд (…), но для последней итерации формула будет не такая. Так как вероятность ни коим образом не означает, что элемент гарантированно появится, то туда же нужно добавить случай неуспешного поиска (т.к. кол-во итераций при нахождении элемента в самом конце и при неудаче одно и то же), поэтому это 1 – вся сумма. Понятное дело, что вся сумма вместе с итерацией N будет равна 1 (полная вероятность, мы учли все возможные случаи); однако нас больше интересует другое наблюдение. Посчитая мат. ожидание вот такой величины i (iteration), мы получим ее среднее значение, т.е. как раз средний случай, сколько нужно выполнить итераций при данных N и p. Для мат. Ожидания необходимо просуммировать произведения величин iteration на ее вероятности, и в результате приходим вот к такой “замечательной” формуле:

Здесь мы просто домножили первый ряд на (k+1) как номер итерации, а последнее слагаемое на N. Не очень красиво, да, но с помощью упорства “математика”, прямых ручек (и немного читерства из интернета, которое может онлайн посчитать сумму любого степенного ряда), получаем в конце концов вот такое значение:



 Ну, а если это упростить еще (что я почему-то не сразу заметил), то получится вообще шикарно:

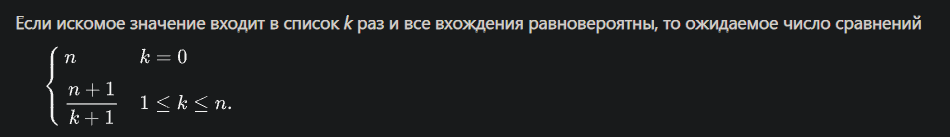
Таким образом, получилась красивая формула ***среднего значения итерации первой встречи элемента в массиве размера N при вероятности p его встречи в любой из ячеек.***

*Примечание*: наверняка эту же формулу можно получить намного легче, если даже не аналитически:

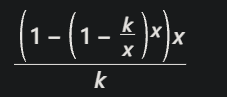
p – вероятность встречи таргета в ячейке; => (1 – p) – вероятность не встретить таргет в ячейке; => степень – вероятность не встретить таргет в N ячейках подряд (т.к. события встречи независимые, то мы можем просто их перемножать); => (1 – степень) вероятность *ТОЧНО* встретить таргет хотя бы раз в N ячейках подряд; далее 1/p – это *концентрация* таргетов в массиве, то есть на сколько элементов в среднем в массиве приходится одно появление таргета; ну а чтобы получить среднюю позицию первой встречи таргета, нужно эти две величины (концентрацию и вероятность точно встретить таргет хоть где-то) просто перемножить.

И да, черт возьми, комбинаторно посчитать это намного проще, чем час мусолить мои ряды :) Однако не видя готовой формулы лично мне до такого было сложно догадаться (или догадаться, что это действительно так работает), а вот обьяснить ее же потом еще и с комбинаторной точки зрения, оказалось действительно очень просто.

Теперь давайте ее проанализируем:

Во-первых, в Википедии есть формула ожидаемого числа сравнений для линейного поиска:

Не очень похоже на нашу полученную формулу, не так ли? Даже больше, они дают разные значения, чего по идее быть не должно. Однако, и эта формула, и наша, выдает дробное значение, где полученные значения и различаются, но при их округлении в целой части они всегда совпадают (проверено на N = 1,…, 10, 100, 1000 и т.д.) при условии, что наше p, это их k/n.

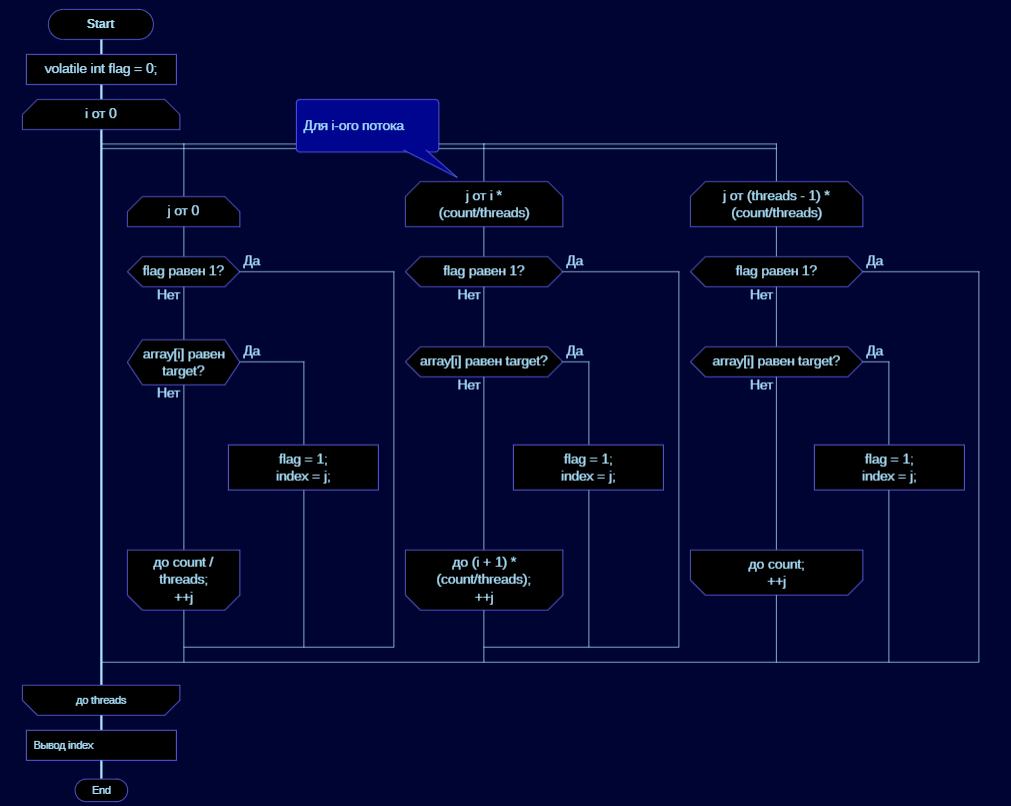
Правда, есть еще одна загвоздка: в их случае мы можем сказать по ожидаемому числу сравнений, что временная сложность алгоритма O(n) (что и так с самого начала понятно), а по нашей формуле мы то же самое указать не можем. Действительно, в знаменателе константа, а вот в числителе 1 – какое-то близкое к единице, но меньшее нее число в степени N. Мы даже не можем сказать, что это степенная сложность. Одна-а-ако, после замены p на k/n получим:

И в пределе x -> oo мы получим, что скобка стремится чисто к 1, то есть сложность так же будет O(n)

Для еще большей наглядности схожести этих двух формул, вместе с отчетом должна быть приложена гифка (compare two lines), где представлено изменение этих двух графиков для промежутка k = (0, 10]. На ней наглядно видно, что они отличаются очень слабо, и это отличие не меняет целой части значения, хотя точность и изменяется при изменении k, причем нелинейно.

Вывод: временная сложность, как и ожидалось, O(n), однако кроме этого была получена формула без приближения исходя из простых основ комбинаторики и теории вероятности, по которой можно менять и параметр размера массива, и концентрации элементов в нем (что нам сейчас и пригодится для правильного проектирования экспериментальных массивов в выборке)

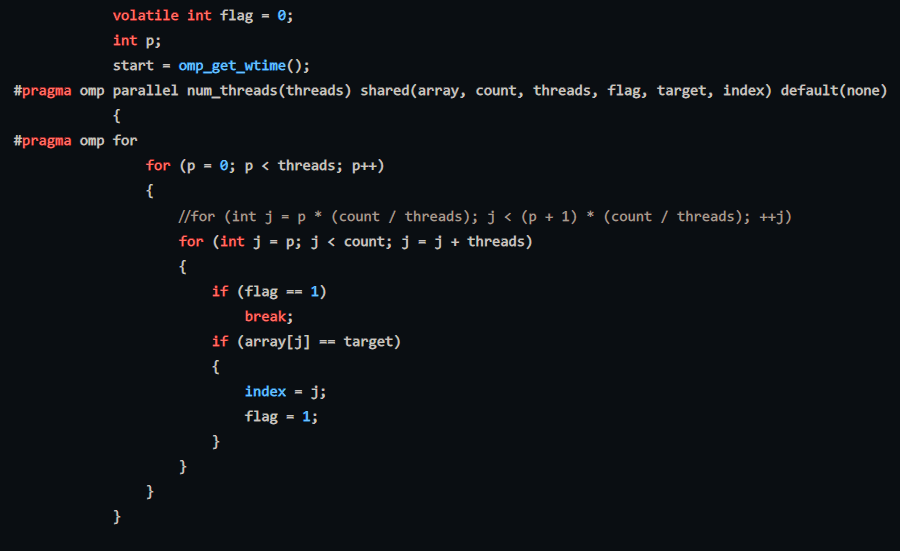
***Реализация параллельного алгоритма***

Проблема распараллеливания линейного поиска состоит в том, что мы не можем использовать оператор break в директиве #pragma omp for, однако никто не мешает использовать его во внутренных циклах, включенных в распаралелленный for. Отсюда возникает идея сделать такой своеобразный костыль. Что мы делали в первой лабораторной? Использовали omp for чтобы разделить весь массив на чанки, а потом параллельно выполнять поиск максимума в каждом из чанков. Здесь же мы сделаем это разделение вручную: omp for мы передадим цикл, в котором каждый поток возьмет на себя одну единственную итерацию, и в зависимости от номера итерации уже *ВНУТРЕННИЙ* цикл for будет выполнять поиск в нужном чанке. Это позволит нам использовать break, а чтобы передать информацию о нахождении элемента каким-либо потоком, можно будет использовать какой-нибудь флаг, помеченный как volatile, чтобы обеспечить исключительное изменение флага. Таким образом, мы приходим к следующей блок-схеме:

Соответственно, как только какой-либо поток поменяет состояние флага на true (1), все остальные потоки получат информацию об этом либо на текущей, либо гарантированно на следующей итерации, и тоже закончат свою работу.

*Примечание*: в OpenMP специально для этого есть специальная директива #pragma omp cancel (cancellation point). Однако, по сути, она работает так же, как эта реализация, но работает в основном медленнее, чем ручная генерация чанков и реализация вложенного цикла.

Таким образом, сама реализация алгоритма выглядит следующим образом:



***Временная сложность параллельного алгоритма***

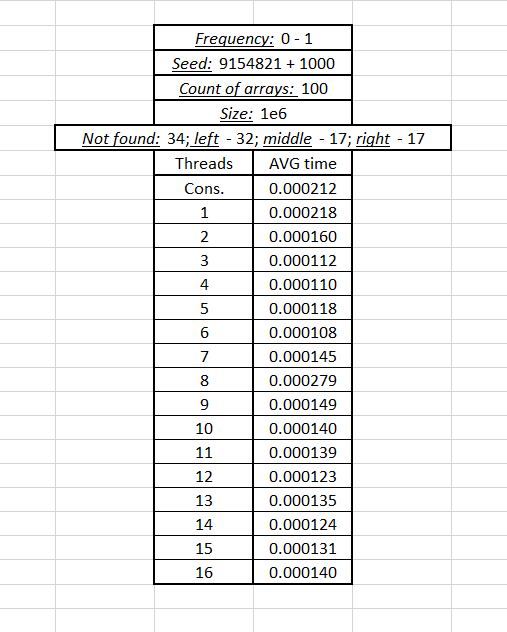
В этом случае оценить временную сложность в среднем случае достаточно сложно, т.к. при таком разбиении массива на чанки от расположения таргета очень сильно зависит и время работы алгоритма: если все таргеты расположены в конце чанков при одном кол-ве процессоров p, то поиск будет идти за O(N/p), а при другом значении p эти же элементы (или хотя бы один из таргетов) могут оказаться в начале чанков, и тогда поиск займет один временной такт. Поэтому тут нужно ввести дополнительную переменную окружения, как связь кол-ва процессоров с окружением, но это уже достаточно нетривиальная задача, и я не знаю, как ее можно решить. Однако в среднем, можно предположить, что он будет так же работать в p раз быстрее, так как мы снова так же, как и в первой лабораторной, делим массив на p частей. Но в нотации O-Big это все так же останется линейная сложность.

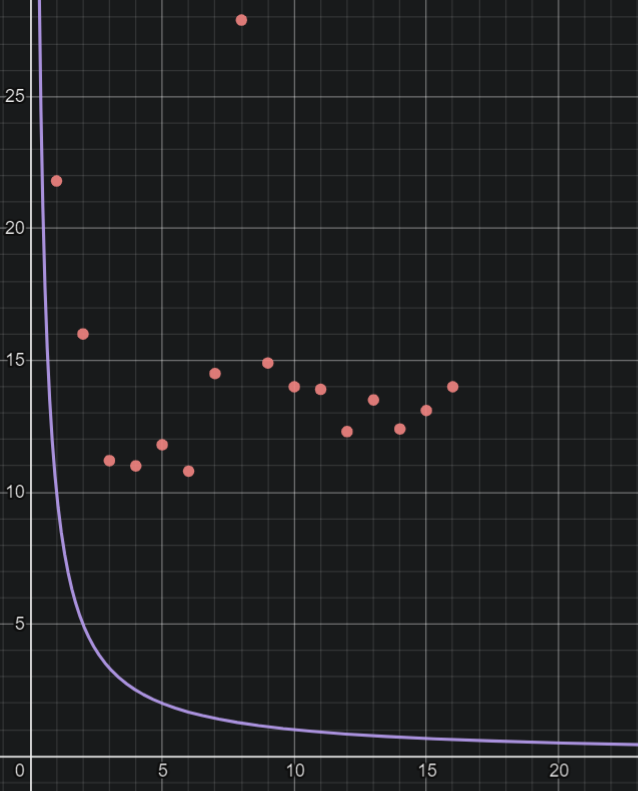
***Экспериментальные данные***

Как мы поняли, от частота встречи искомого элемента, вообще говоря, зависит очень многое. Однако на практике сложно себе представить ситуацию, когда данная вероятность вообще известна (действительно, зачем нам поиск, если чтобы оценить его эффективность нам нужно им воспользоваться, чтобы найти все нужные элементы?). Однако мы предположим, что нам известна эта частота (пер. frequency в коде), а также чтобы измерить эффективность в среднем случае, выборка будет идти не от 10, и даже не от 100, а из 1000 (кроме случая слишком большого размера массива, ибо программа может считать это часами) различных между собой, но одинаковых массивов. А чтобы еще больше убедиться в работоспособности алгоритма, возьмем эти массивы не одной блины, а хотя бы трех различных: 1e6, 1e7, 1e8. Таким образом мы хотя бы приблизительно усредним все возможные случаи, когда к примеру с частотой frequency = 1 элемент в массиве вообще не встречается, а где-то встречается несколько раз, тем самым увеличивая скорость его поиска считая от начала текущего для данного потока чанка.

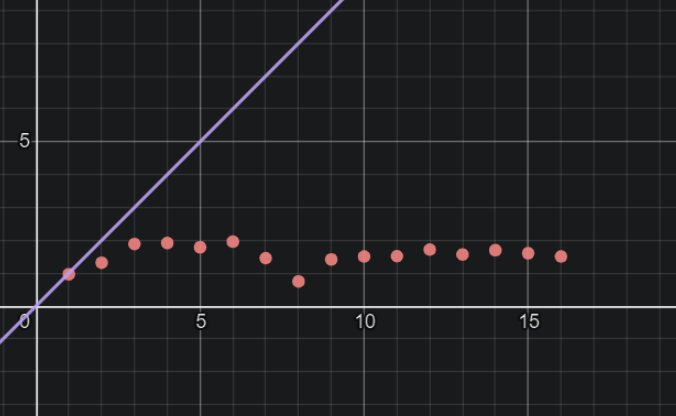
Кроме того, после каждого запуска выводится дополнительная информация о том, где конкретно этот элемент был найден при *ПОСЛЕДОВАТЛЬНОМ* запуске, чтобы примерно понимать статистику: left – первая треть массива; middle – середина; right – конец (последняя треть) массива; ну и not found – обозначает, что поиск прошел неудачно.

Итак, будем последовательно представлять время работы для последовательного и параллельного алгоритма для конкретного значения frequency и count (размер массива):

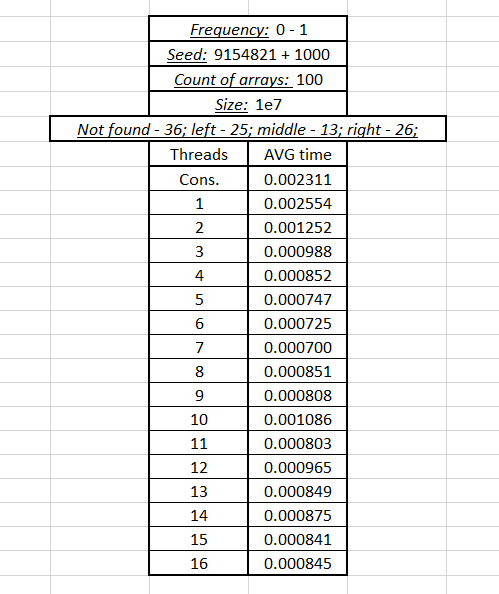
* *frequency = 1, count = 1e6*:

 *Время:*

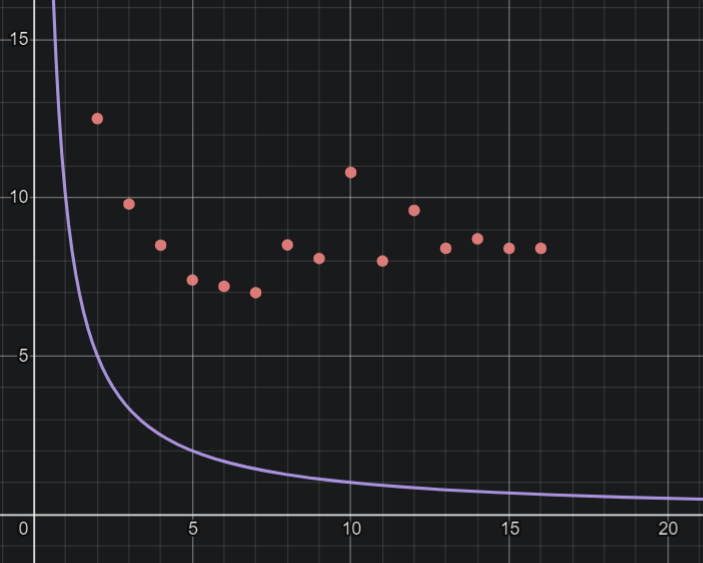
*Ускорение: Эффективность:*

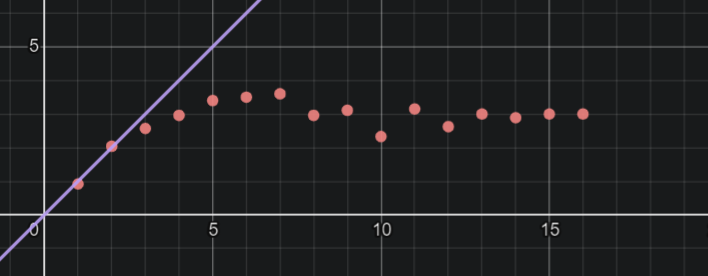


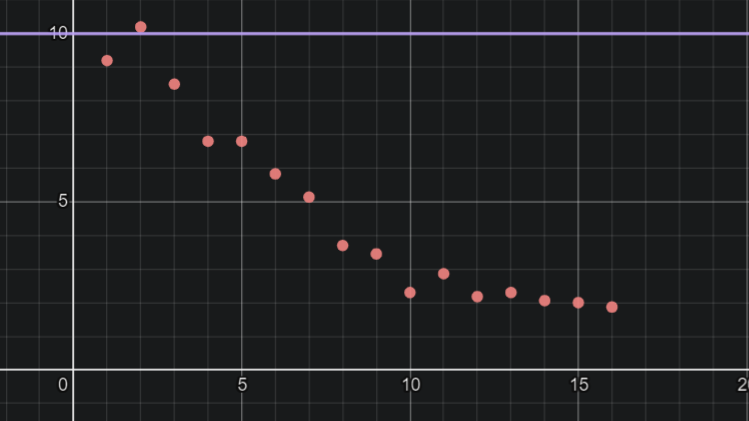
Как мы видим, с теоретическими предположениями практика совпадает буквально до 3 процессоров, а потом начинается третий акт Лебединого Озера. В том смысле, что значения, даже при усреднении каждого из поисков в массиве по 10 опытам, скачут как хотят, не заморачиваясь особо о том, что им нужно ускоряться из раза в раз. В общем, им параллельно :) А при восьми процессорах вообще происходит аномальное падение производительности (что, кстати, будет так же заметно и в других опытах)

** Будем продолжать увеличивать размер массивов на порядки:

* *frequency = 1, count = 1e7*:

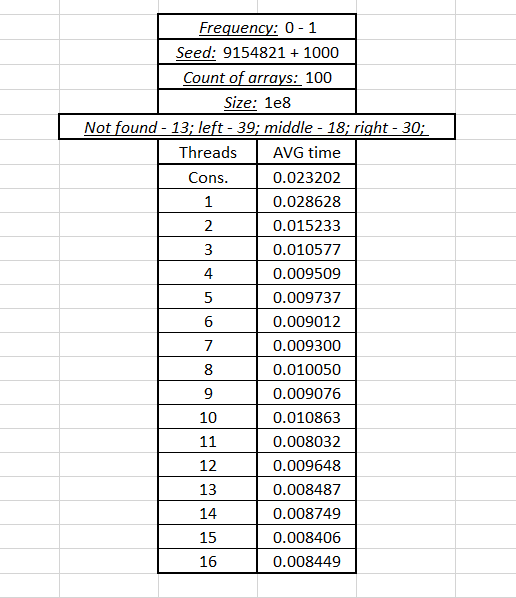
 *Время: Ускорение:*



 *Эффективность:*

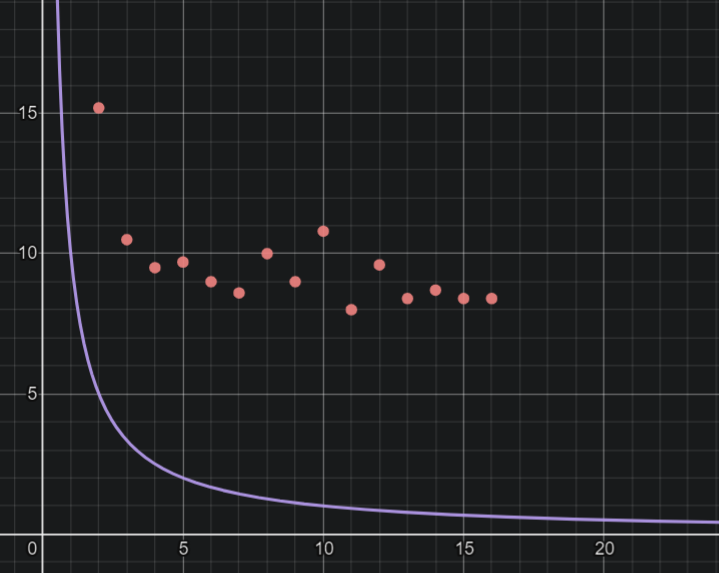
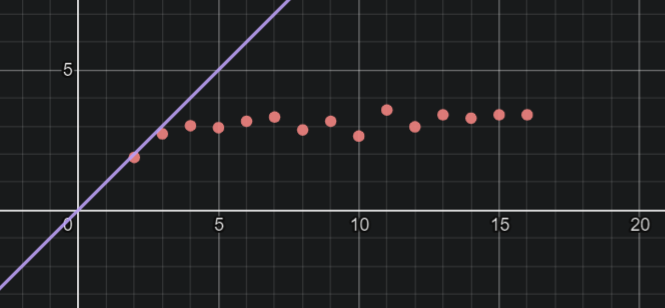
Уже лучше: до 7 процессоров включительно ускорение перестает прыгать и выдает приемлемые результаты; и максимальное ускорение достигает ~ 3.3, что уже не плохо, хотя и далеко от теории.

Увеличим размер еще на порядок:

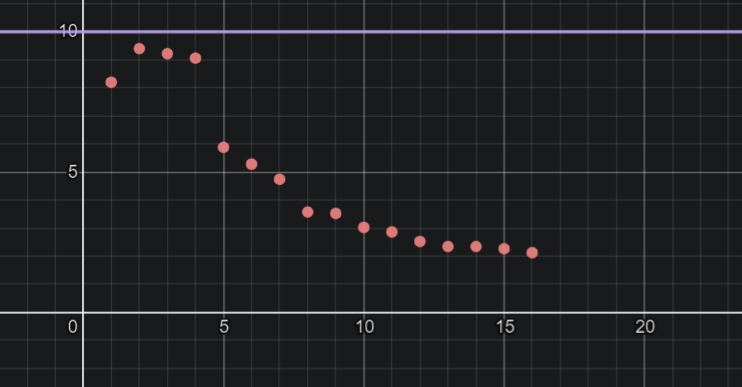


* *frequency = 1, count = 1e8:*

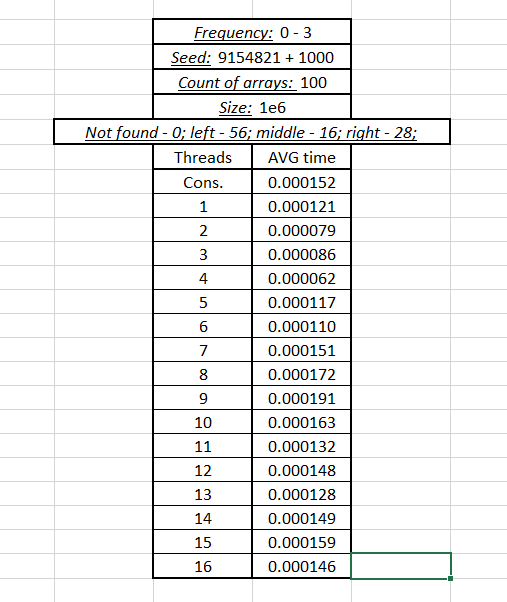
*Время: Ускорение:*



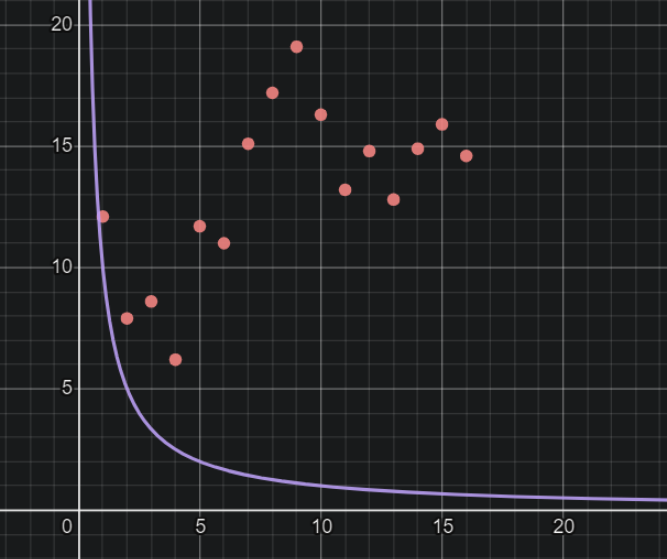
*Эффективность:*

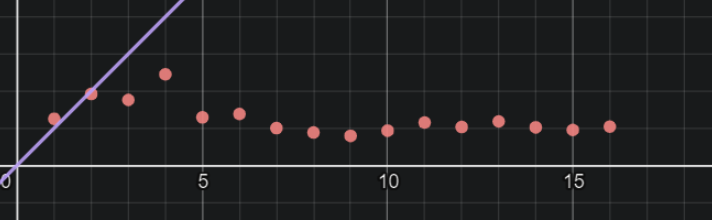
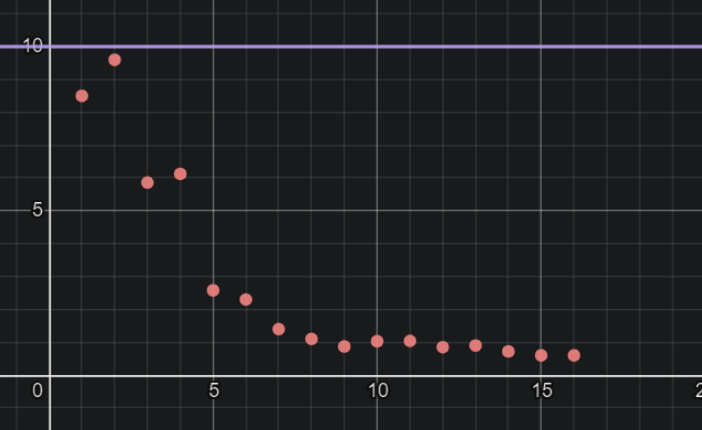


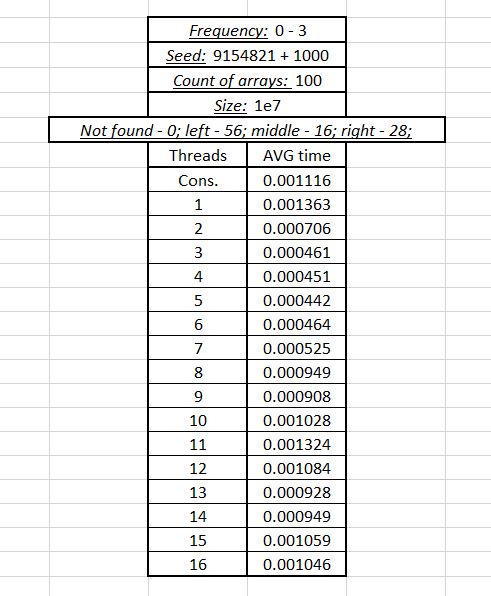
Здесь результаты самые адекватные из всех трех опытов, хотя падение эффективности после 4-ех процессоров существенно. Мы сумели добиться ускорения более чем в 3.5 раза.

 Однако во всех трех случаях у нас частота таргета была равна единице; то есть в среднем в массиве был один элемент. А что будет, если увеличить это число, скажем, до 3 вхождений? Теоретически, эффективность должна возрасти. Но так ли это?

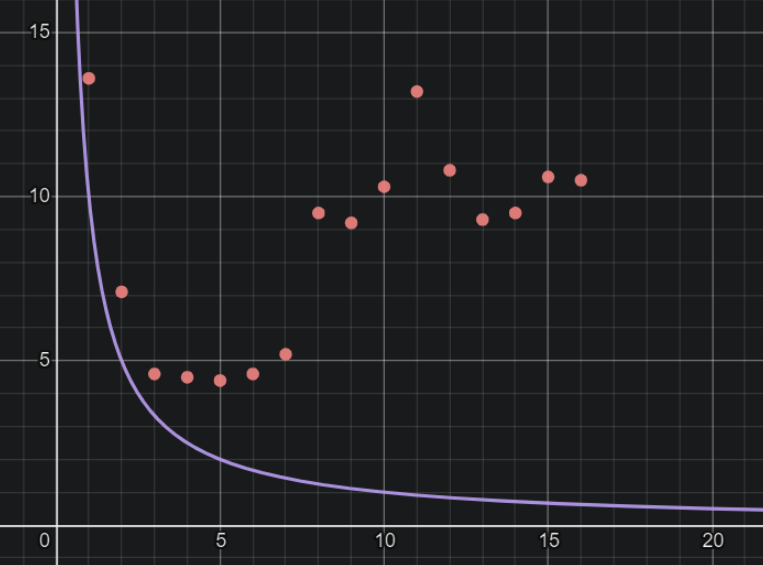
* *frequency = 3, count = 1e6:*

 *Время:*

 *Ускорение: Эффективность:*

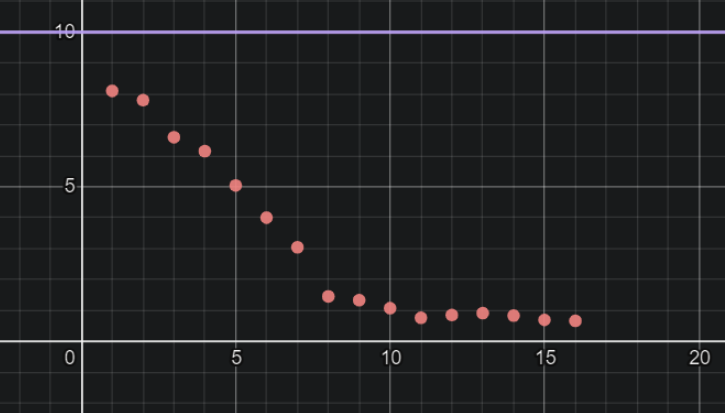
 Все такое же… Разве что максимальное ускорение возросло до ~ 2.5 (при 4 процессорах). Идем дальше:

* *frequency = 3, count = 1e7:*

 *Время:*

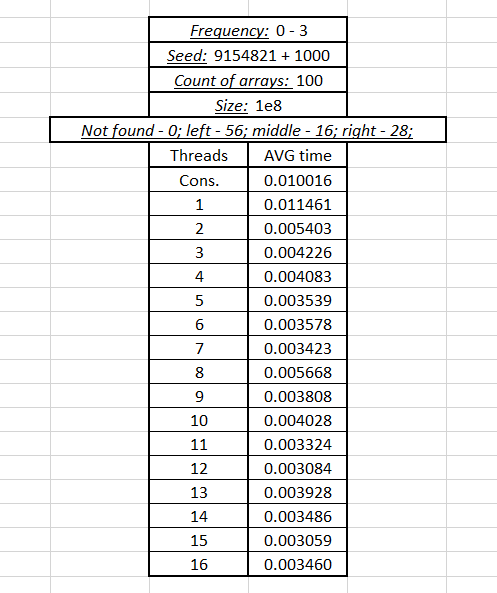


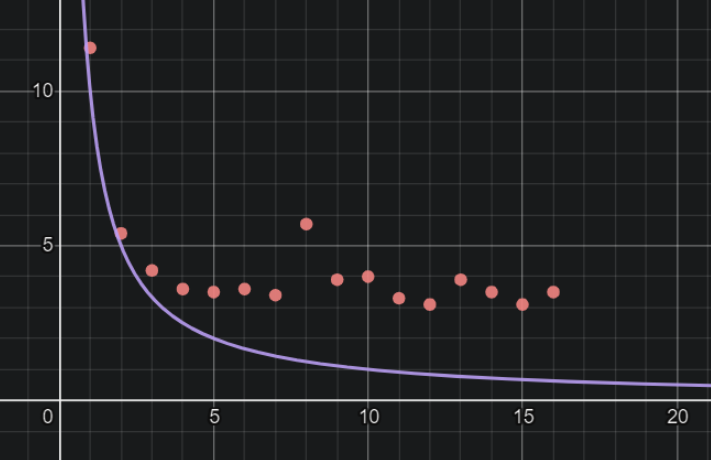
*Ускорение:*

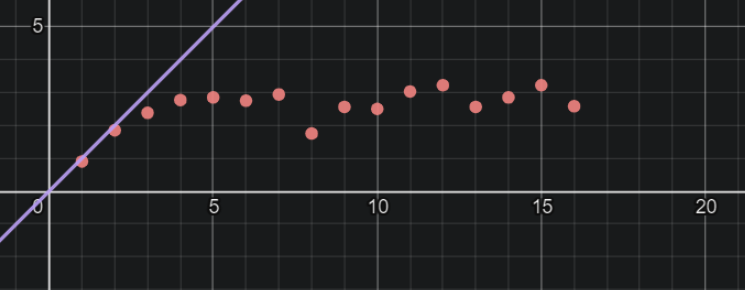


*Эффективность:*

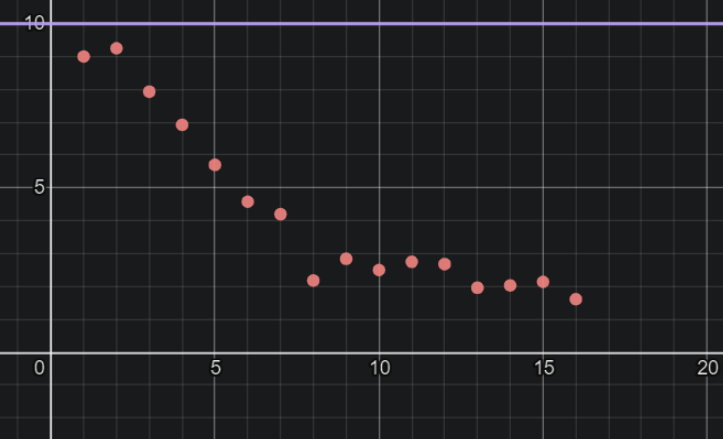
Тоже не очень хорошо, даже хуже, чем при frequency = 1. Однако, остается еще размер 1e8:

* *frequency = 3, count = 1e8:*

 *Время:*



*Ускорение:*

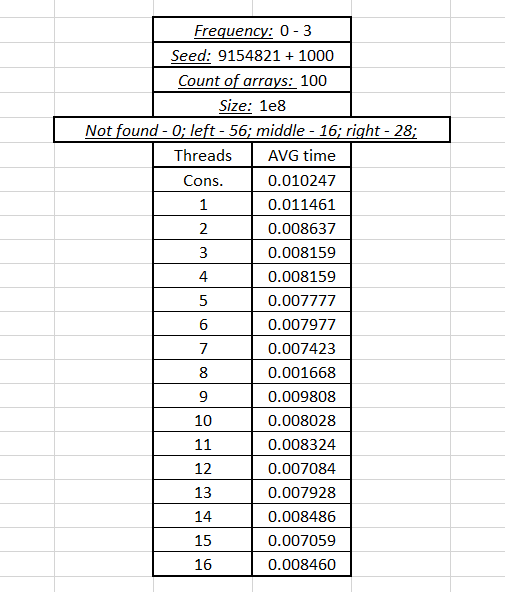


*Эффективность:*

Итого: нам не удалось получить ускорения больше, чем в 3.5 раза, и в реальности выделять процессоров больше четырех смысла нет, так как в сравнению с выигрышем в скорости траты ресурсов более критичны.

***Оптимизация алгоритма:***

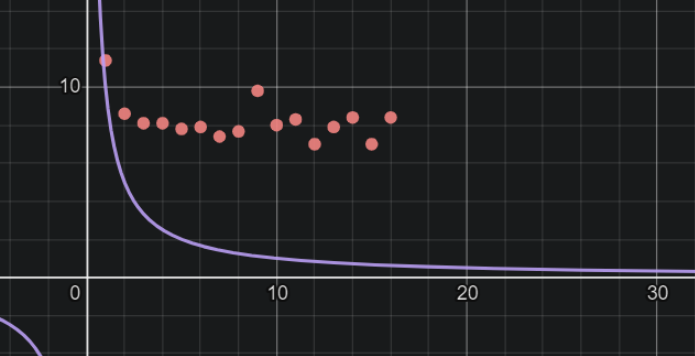
Однако есть еще кое-что, что поможет нам ускориться. Если вспомнить реализацию нашего алгоритма, то мы разделяли весь массив на p примерно равных чанков, идущих друг за другом. Однако из-за этого получить среднюю оценку очень трудно по оговоренным ранее причинам: нужна еще одна переменная окружения, связывающая расположение элемента с кол-вом процессоров. Однако, что нам мешает распределить элементы немного по-другому? В OpenMP есть специальный параметр для распараллеленных циклов: schedule(method). method по умолчанию стоит в значении “static”, что означает, что процессоры будут делить цикл на p равных частей, следующих друг за другом. Но method может принимать и другие значения, например, dynamic. В таком случае процессоры поделят индексы между собой по правилу i; i + threads. То есть мастер возьмет себе элементы 0, threads, 2 threads и так далее, первая сгенерированная нить 1, 1 + threads, 1 + 2 threads…. Таким образом, мы каждой новой итерацией обрабатываем threads подряд идущих элементов. Что нам это дает? На самом деле, очень многое, ибо при последовательном поиске мы просто в лоб идем по массиву, пока не встречаем таргет. А при таком разбиении в параллельной реализации с dynamic schedule мы точно так же идем по массиву, но в threads раз быстрее. То есть мы можем здесь сказать, что временная сложность будет O(n/threads).

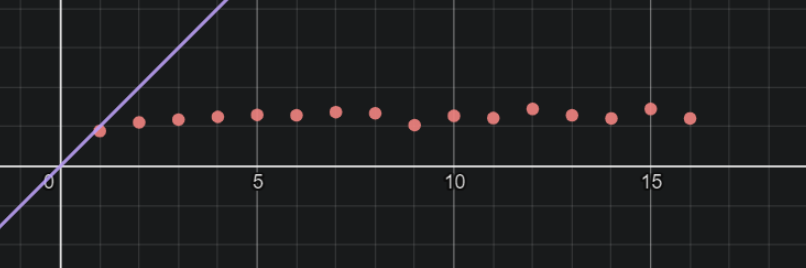
Давайте проверим эту теорию на практике (чтобы реализовать данное разбиение, достаточно раскомментировать 71 строку кода и закомментировать предыдущий for):

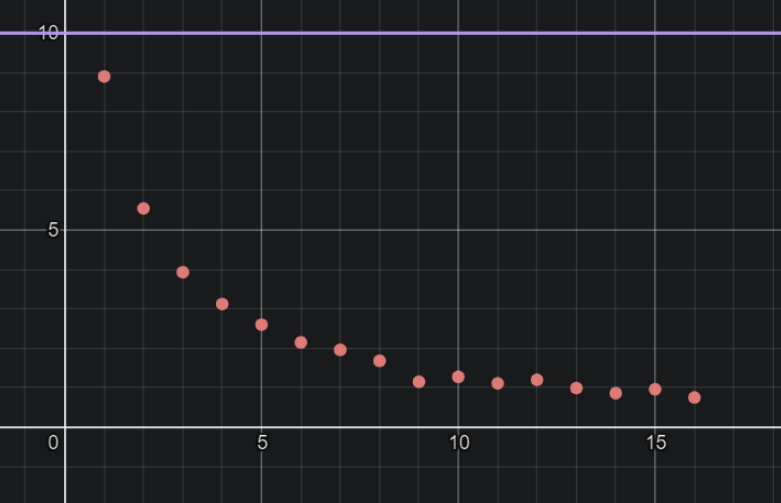
* *frequency = 3, count = 1e8:*

*schedule(dynamic)*

*Время:*



*Ускорение:*



*Эффективность:*

И получаем мы … А ничего мы не получаем, оно не работает. Максимальное ускорение, которое мне вообще удавалось получить, это 2. Я проводил эксперименты со всеми теми же данными, что были представлены выше, но результат один и тот же: ускорение почти не меняется, оно практически константа, варьируясь от 1 до 2. И я не понимаю, как это можно объяснить, ибо оно ***ДОЛЖНО***, просто ***ОБЯЗАНО*** работать быстрее, но видимо, я что-то не понимаю в механизмах OpenMP.

***Выводы***

В результате анализа линейного поиска была получена формула, которая может быть использована для определения средней величины необходимого кол-ва итераций для успеха, а также сравнение ее с общедоступной формулой.

При проектировке выборки были учтены полученные при анализе линейного поиска результаты и усреднены массивы таким образом, чтобы покрыть максимально возможное количество вариантов, и проведено 6 основных опытов для 2 различных значений частоты и размера массивов. Однако не смотря на различные входные данные, их усреднение, ускорения более чем в 3.5 раза получить не удалось.