1. Общие характеристики виртуальной машины:

Система – Red Hat (64bit) («виртуалка» на Oracle VM с выделенными ~2ГБ оперативной памяти).

Всего доступной оперативной памяти 1.8ГБ в ненагруженной системе, в действительности же со всеми запущенными процессами свободной памяти к моменту запусков скриптов будет около 1.65ГБ.

На ядра лимита не установлено.

1. Общие характеристики хостового компьютера:

Процессор AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz (8 ядер).

8ГБ оперативной памяти (3 уровня кэшей).

Поддержка виртуализации включена.

1. Общий план экспериментов с детализацией:

Иерархия запускаемых файлов, следующая:

1. Базовый скрипт, реализующий алгоритм.
2. Промежуточный скрипт (starter<name>.sh), запускающий базовый скрипт на нескольких разных значениях (файлах).
3. Следящий скрипт (watcherScript<>.sh), запускающий промежуточный скрипт несколько раз на нескольких разных N от 1 до 20, вычисляя среднее арифметическое значение времени реального времени исполнения при помощи утилиты /bin/time.

Соответственно план оставлен без особых изменений – выбираем кол-во ядер (1 или 2), выбираем параллельное или не параллельное исполнение – запускаем следящий скрипт.

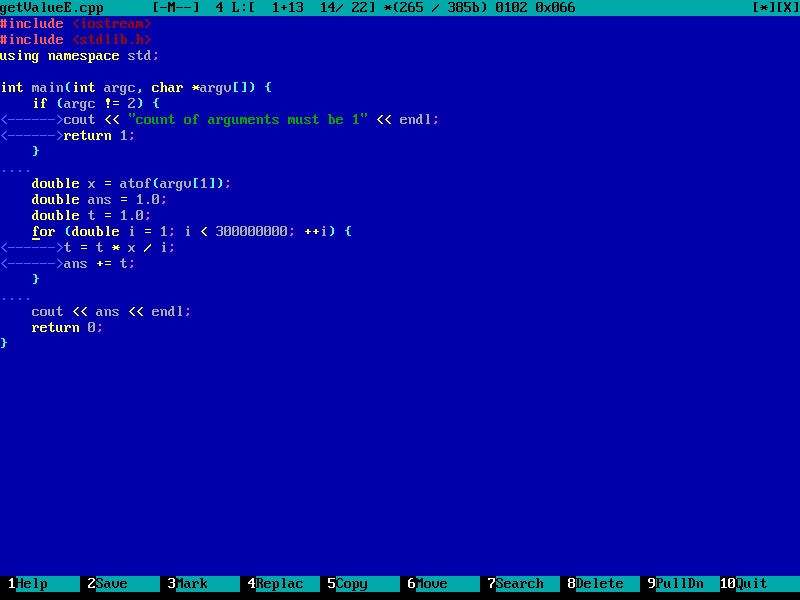
Таким образом считаем средний real\_time (%e) исполнения процессов при разных N и фиксируем это на соответствующих графиках.

1. Алгоритмы:

**Алгоритм 1):**

(Реализация на с++)

Принимает ровно одним аргументом x. Считает с высокой точностью f(x) = e^x (по ряду Тейлора). В вычислительном цикле присутствует по одной операции сложения, умножения, деления и присваивания. В конце происходит одинарный вывод в консоль итоговой суммы ряда.



getValueE.cpp (Базовый скрипт)

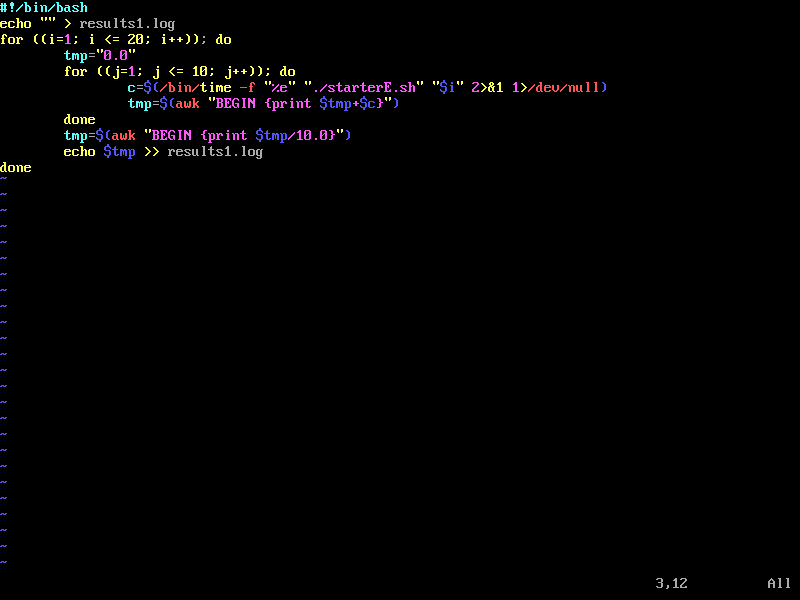
То есть затраты на работу с вводом/выводом у нас минимальны, а в основном выполняются присваивания и арифметические операции.

Пороговым значением для работы около 2 секунд у меня оказалось примерно 300000000 операций в цикле. (компилирую на -O0 без дополнительных флагов для лучшей показательности)



starterE.sh (Промежуточный скрипт)

Для параллельной версии просто запускаем все в фоне (с помощью &) и ставим команду wait после цикла, чтобы time зафиксировал время при окончании исполнения всех скриптов.



watcherScript1.sh (Следящий скрипт)

График (T1 – запуск с последовательным исполнением на одном ядре, T1\_parallel – с параллельным на одном ядре, T2 – запуск с последовательным на двух ядрах, T2\_parallel – запуск с параллельным на двух ядрах, Nsys350-400 дополнительный параметр, обнаруженный в ходе экспериментов):

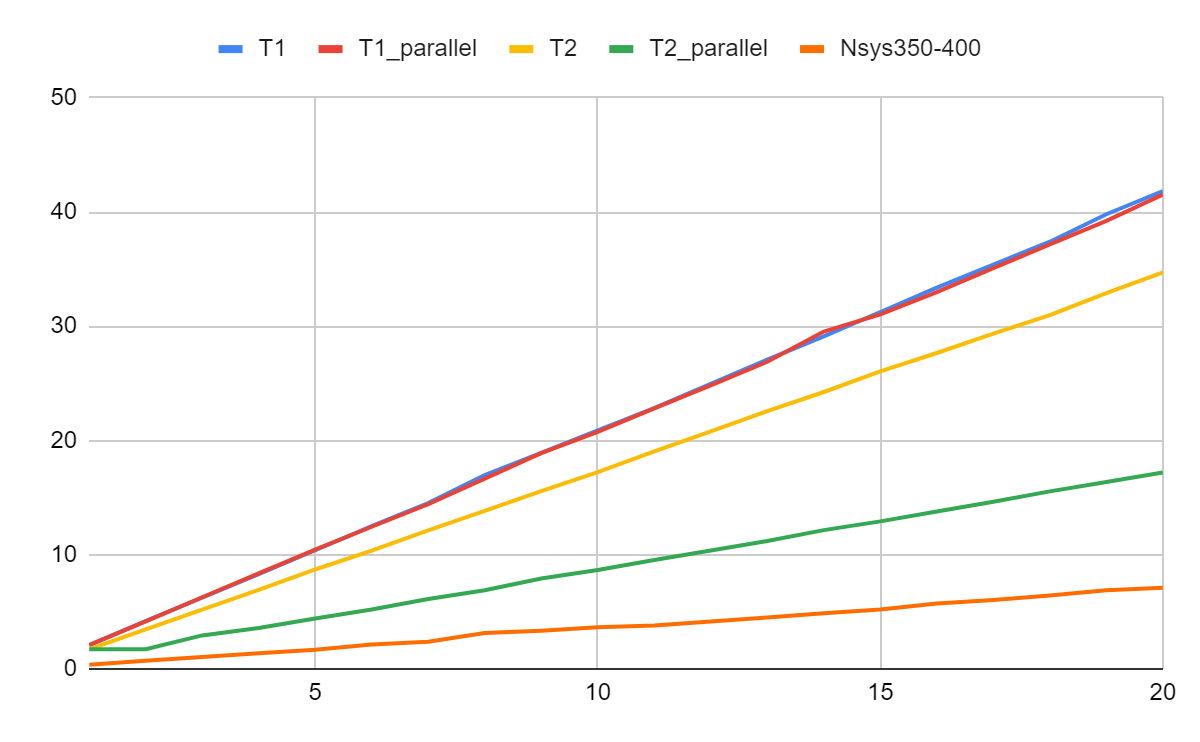


График 1 (Первая серия экспериментов)

**Выводы:**

Видно, что все зависимости от N линейны вне зависимости от ядер и параллельности.

T1 и T1\_parallel совпадают (по понятным причинам, т. к. вне зависимости от параллельности расходуется процессорное время одного и того же процессора)

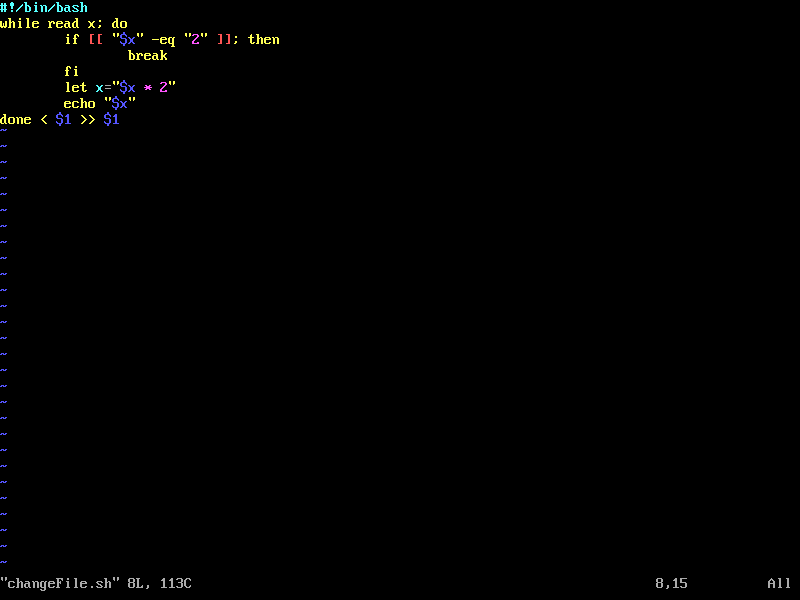
T2 отличается в каждой точке N от T1 примерно на некоторую константу c \* N (ровно на ту самую прямую Nsys350-400). По всей видимости дело в том, что на моей системе независимо от всего ОС использует на каждый вызов скрипта, хоть как-то использующего вывод в консоль (cout в нашем случае), фиксированные 350–400 мс процессорного времени в режиме kernel\_mod (sys); real\_time же складывается из user\_mode (%U) + kernel\_mod(%S) + переключения + ожидания. При 2 же процессорах kernel\_mod берет на себя второй процессор (-350 мс), а также на него могут распределяться стандартные фоновые системные процессы и следящий скрипт (eps мс), из-за чего и происходит данное ускорение. Абсолютно такую же зависимость мы увидим во втором эксперименте.

T2\_parallel логичным образом в каждой точке примерно 2 раза меньше T2 (то есть процессоры с эффективной параллельностью распределяют между собой серии арифметических операций).

**Алгоритм 2):**

(Реализация на bash)

Скрипт принимает имя файла вида (fm{i}, i= {1..20}), в котором написано 200000 единиц (строчка = 2 символа (“1\n”) = 2 байта, то есть в общем файлы по 2 \* 200000 = 4МБ, именно на таком объеме исполнение на мой системе длится 2 секунды). Далее он по одному считывает символ, умножает на 2 и выводит в конец того же файла (открытие файла ровно одно, но при этом нет буферизации). Таким образом арифметических операций минимальное количество (но они есть) и максимальное кол-во обращений к диску. Следящий скрипт обновляет каждый файл после работы с ним (дополнительным скриптом), чтобы не увеличивать их объем и в целом проводить эксперимент в честных условиях.



changeFile.sh (Базовый скрипт)

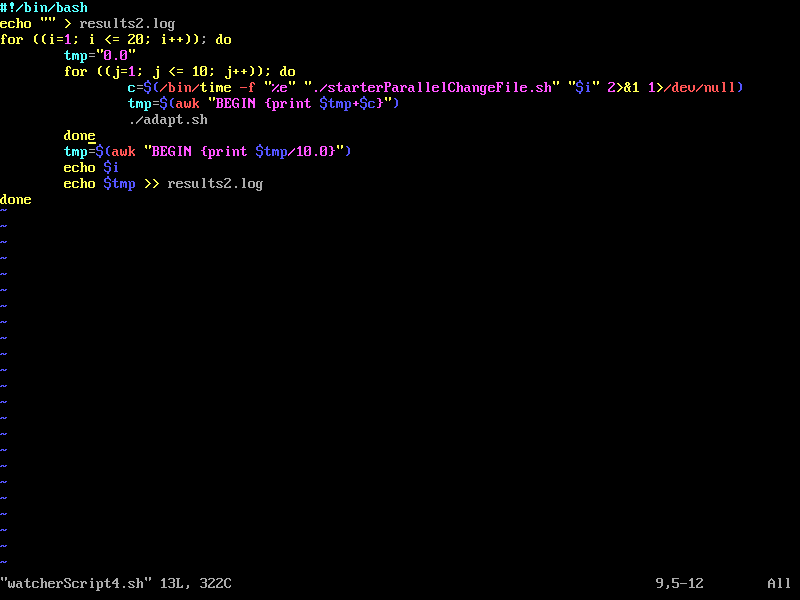
Я реализовал это таким образом что перенаправил стандартный вывод в вывод файл и каждую итерацию read читает, а echo сразу записывает значение.



starterParallelChangeFile (Промежуточный скрипт, версия с параллельным запуском)

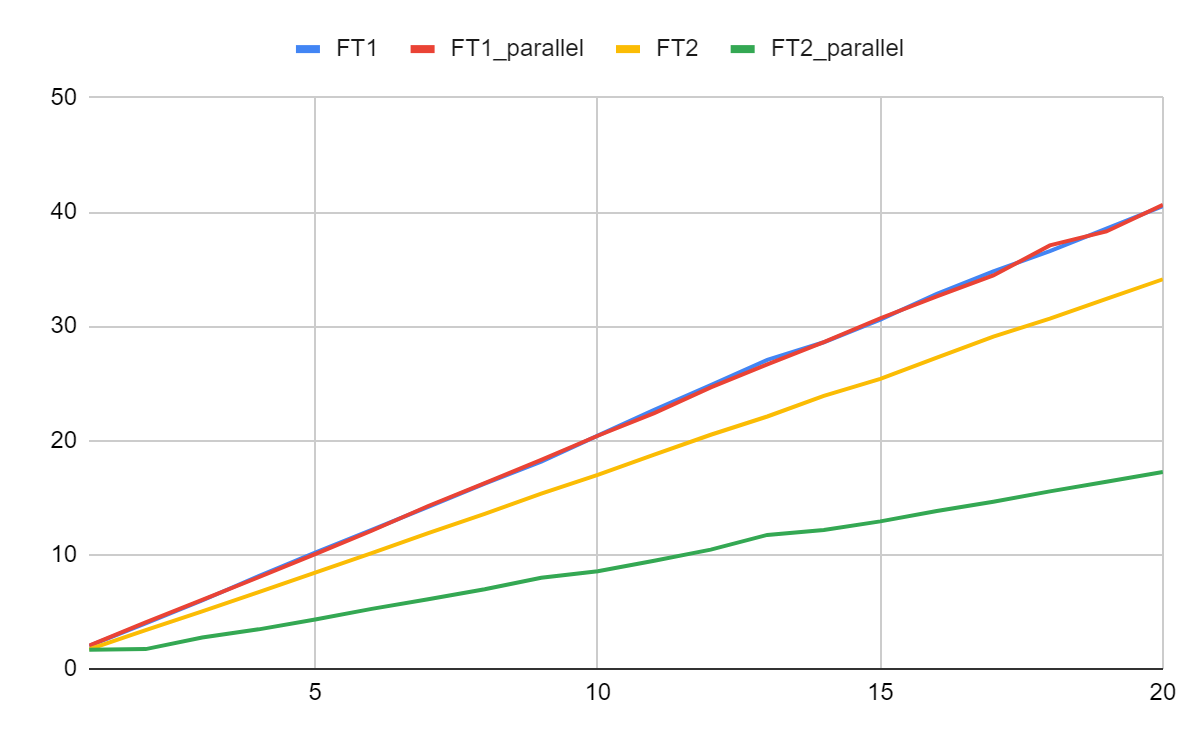


adapt.sh (Дополнительный скрипт для следящего скрипта, чтобы поддерживать условия эксперимента)



watcherScript4.sh (Следящий скрипт)

График (FT1 – запуск с последовательным исполнением на одном ядре, FT1\_parallel – с параллельным на одном ядре, FT2 – запуск с последовательным на двух ядрах, FT2\_parallel – запуск с параллельным на двух ядрах):

График 2 (Вторая серия экспериментов)

**Выводы:**

Аналогичные линейные зависимости (и даже графики почти в точности совпадают с соответствующими в первом эксперименте, т. к. я и там и там подобрал одинарное исполнение базового скрипта в 2 секунды).

FT1 и FT1\_parallel также совпадают.

FT2 все также на cN+eps меньше FT1.

FT2\_parallel также в 2 раза быстрее FT2 (это можно объяснить тем, что хоть ввод/вывод более сложная операция для параллельности, у нас в скрипте помимо тяжелого ввода/вывода на каждой итерации присутствует легкое арифметическое действие, что дает возможность процессорам распределять это очень эффективно и без лишних простоев на конвейере).