

Huffman

Код за решението, графики в отделни файлове, както и csv-та по които са генерирани графики:
https://github.com/god-rosko-vs-the-bugs/Rsa_Project_Rosen.git

В това което съм разбрал о задачата е да се чете файл и в последствие да се обработи прочетеното от файла паралелно за да се образува честотна таблица.

Направени тестове:

генерира се файл с големина 1Гб с ето тази команда : `head -c 1G </dev/urandom >lorem4` (няма особен смисъл за тестване с по-голям файл, само отнема повече време за тестване) и изпълняваме тестове с шел два скрипта. Имплементацията е на С чрез pthread библиотеката .Имплементирани са 2 подхода към задачата които са сходни и се различават в един аспект. Следователно 2-та шел скрипта. Графики се генерират чрез питонски скрипт на който тръбва да му сменя директориятаа в която търси ръчно (може да се направи по генерично и да се задава като параметър но, скрипта се пуска само 4 пъти и без това). Направени са тестове на сървър с от 1 до 32 нишки, както и тестове на моята лична машина отново със същия брой нишки.

Подход: има 3* подхода за решаване на задачата, в останалата част от файла ще се отнасяме към тях с тази им номерация:

- 1) по отделно се чете за всяка нишка в буфер, който после тя чете, като след като приключим да четем минаваме да четем за следващата. Като надеждата е че винаги ще имаме една нишка която да е спряла за да и дадем нов буфер, докато всички останали работят.
- 2) Четем един огромен буфер, който даваме на нишките да четат след като е прочетен, като всяка нишки си има диапазон който тръбва да чете.
- 3) Прочитаем си целия файл в един буфер и след това го обработваме с подход 2. Разбира се това е доста малоумен подход, предвид че много лесно може да ни се даде файл който няма как да го поберем целия в оперативната памет,
- 3.1) mmap-ваме файла, и след това четем от него, като така премахваме проблема с големия размер, но това по себе си също е бавна операция и по оптимална ако не четем последователно а искаме и все пак да работим с файла като с масив и да четем места които не са съвсем едно до друго, за да не местим постоянно къде сме. Този подход не би бил по различен по скорост от четене на няколко странички памет, всеки път като четем, което също го прави безсмислен.

Преди да представя всички графики от данните тръба да спомена че съм добавил още една променлива към това което поема като аргументи. Става дума за променлива дължина на буфера който се чете всеки път при четене от файла. Кое то също оказва ефект , както графиките ще покажат. И по мои наблюдения, освен ако съм разбрал задачата грешно, добавянето на повече нишки само забавя програмата. Проблема не е в обработването на информация, най големия буфер с който може да се чете е 64к което не е малко но в С това не е нищо(и може би в някои от другите езици). Като правих измервания реалното работно време на нишките е 0. 64к итерации не са и хилядна от секундата, и показаното работно време е на практика колко време общо всички нишки чакат на опашки в кернела за разните мутекси които съм ползвал. И бих спорил с помош на данните които съм извадил, че за съответната задача колкото повече нишки добавяме толкова по бавни ставаме, и толкова повече процесорно верме губим (което реално не е загубено, този процес само чака и чете), и би било по продуктивно да четем с една нишка с голям буфер, и

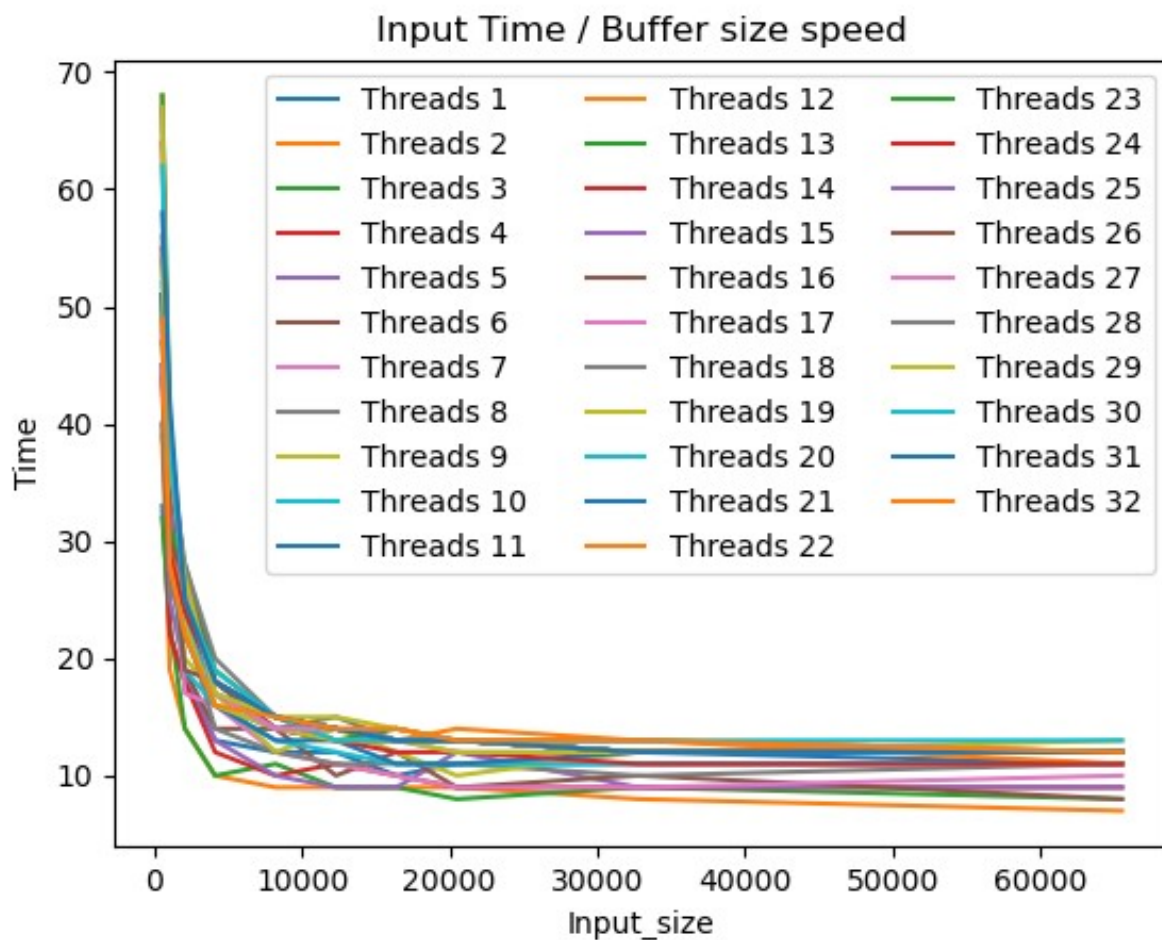
ако добавяме нишки то те би трябвало да четат някой различен файл за максимизация на ефикасността. Проблемата е че не можем да захранваме буферите достатъчно бързо че те да са ефективни. Да, през повечето време спят, което не е нещо крадящо процесорно време но ние не можем да ги храним и без това достатъчно бързо и ефективно въпреки че имаме 32 нишки, на практика работят 2-3 от тях не повече, при това и с най големия възможен буфер за четене, като го намаляме, ефективността става толкова зле че е по бързо да използваме само една нишка която да обработва . Четенето от диска е бавната операция която няма как да забързаме и да искаме. Според мен лоша задача за паралелизация след като става по лошо колкото повече нишки се добавят. И както моите графики ще покажат, иамаме забързване само при увеличние на големината на буфера

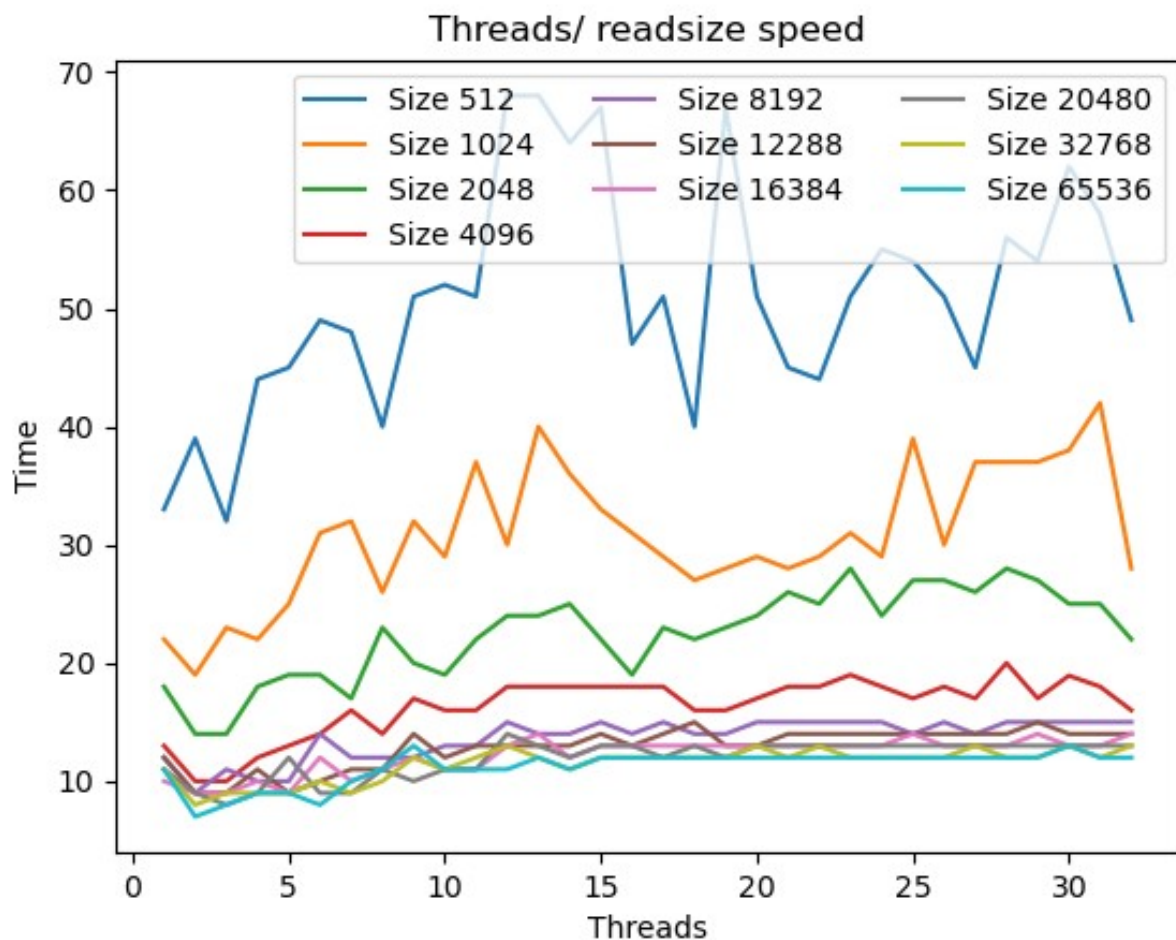
Най важни графики от тестовете:

(статистика за индивидуалните нишки се намира в гитхъб линка, просто не са толкова интересни за доклада)

Тестове от сървъра

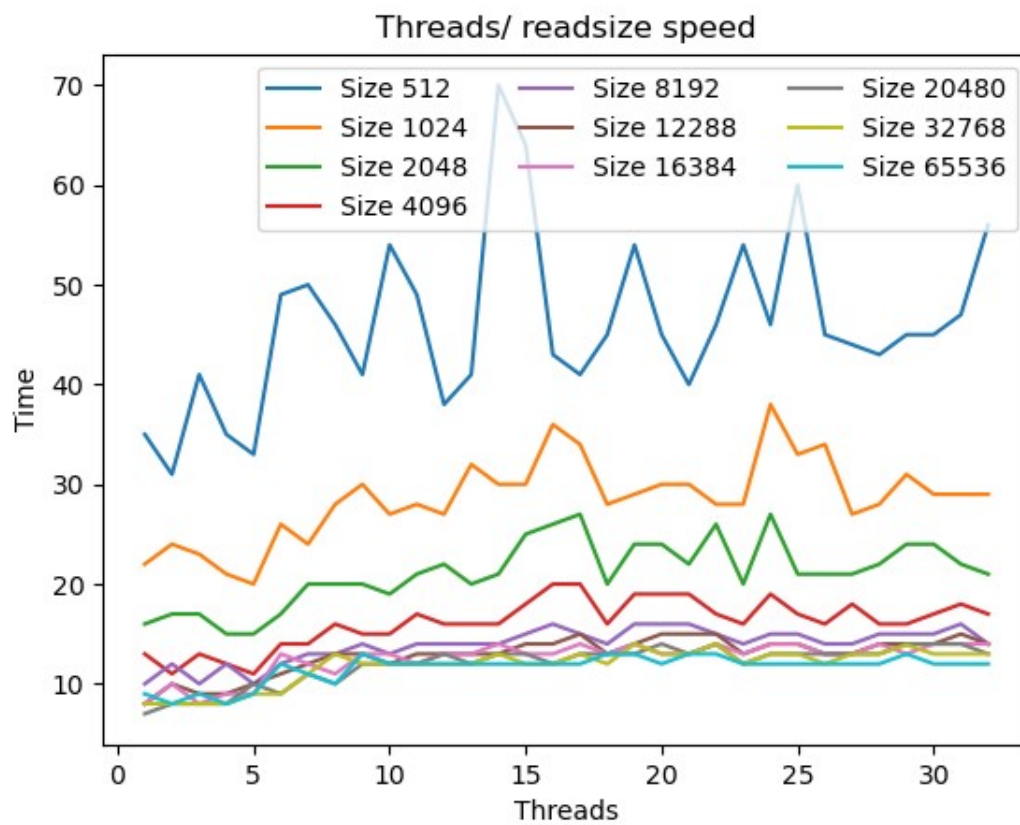
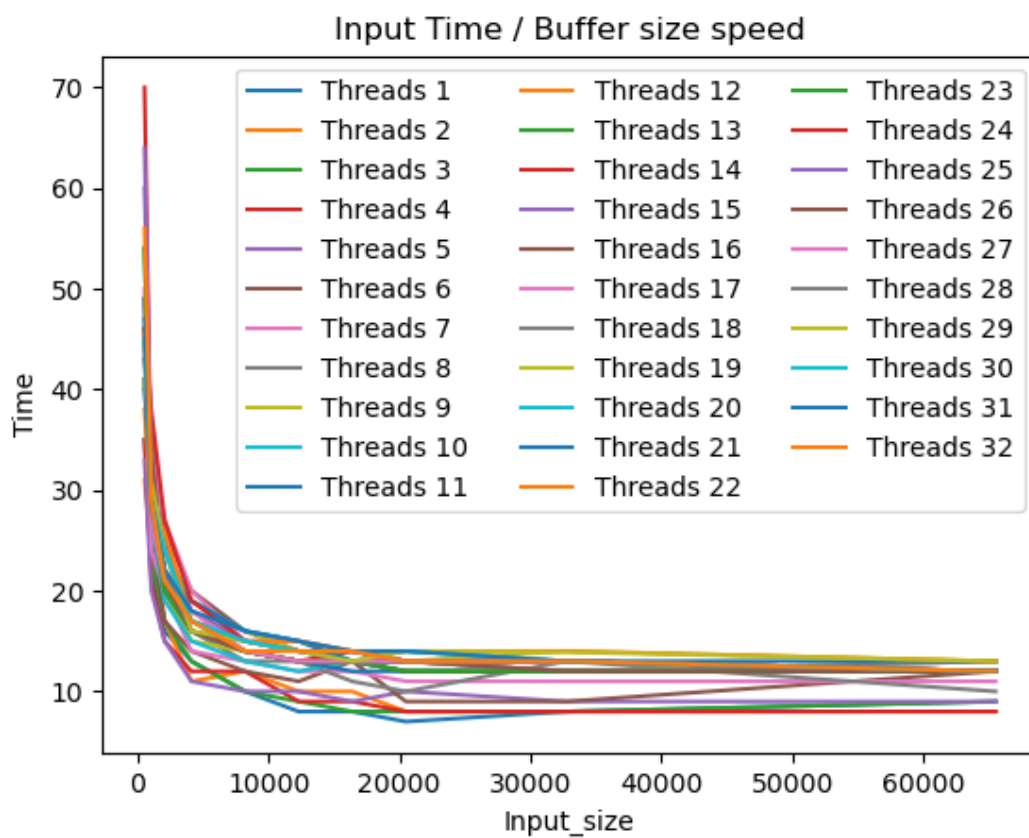
Подход 1:






както се вижда (от 1) колкото по голям буфера толкова по малко времето (четем по рядко)
 и има разлика от 5-10 милисекунди между всички в края. И имат една и съща графика общо взето
 => повече треди != повече ефикасност
 и от втората графика се показва че колкото повече треди имаме толкова по високо е средното
 време.

Подход 2: Същото заключение

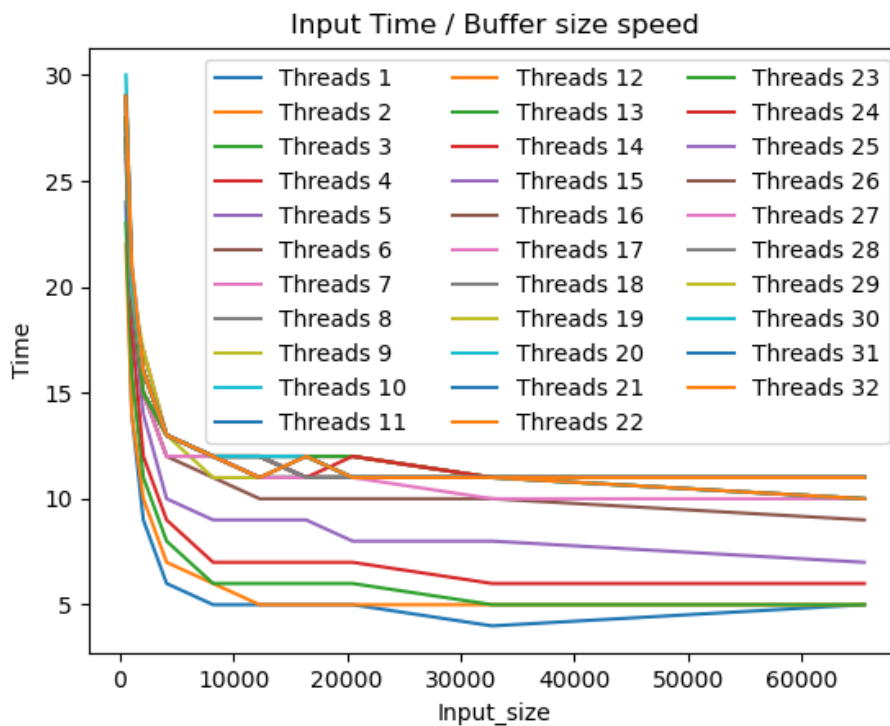


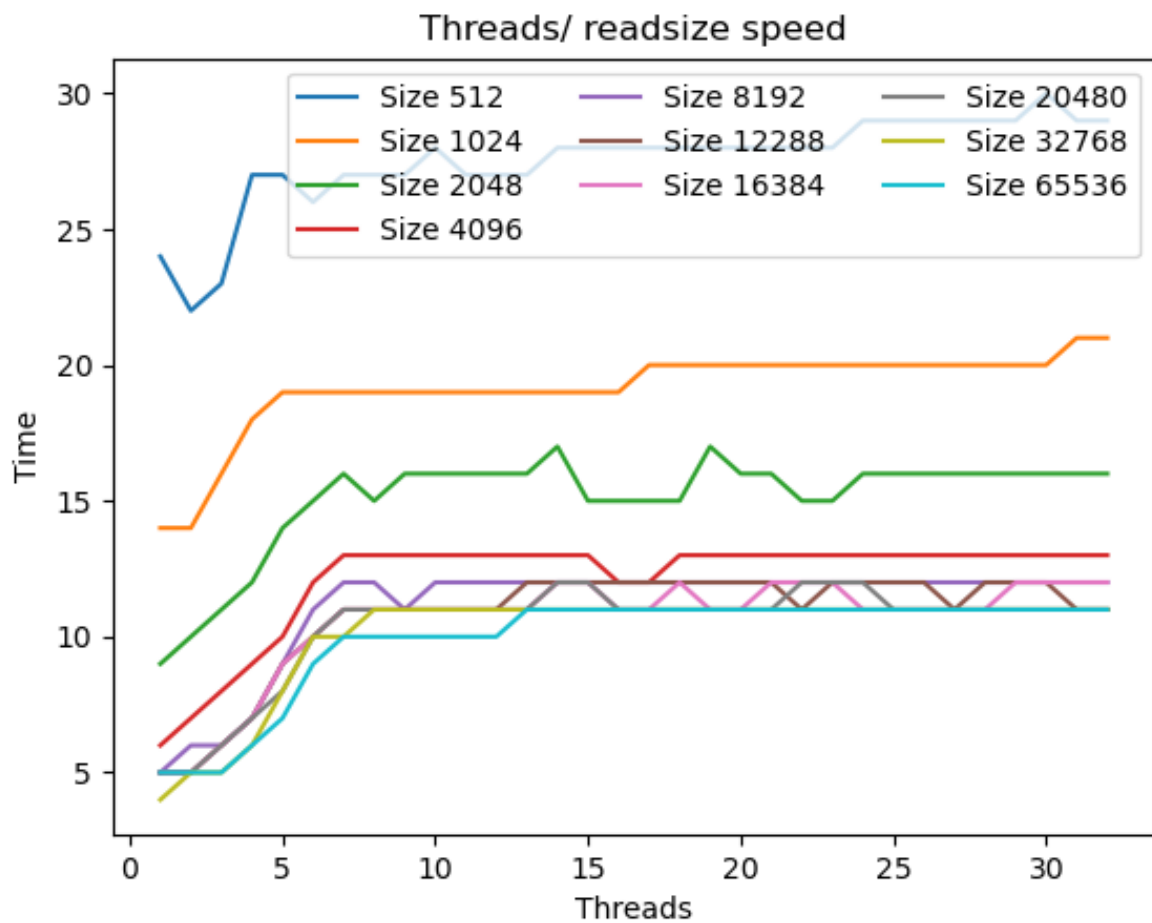
Спецификации на моята мшина: с която съм правил контролни тестове, след като не вярвам на сървъра. Има по малко хардуерни нишки но резултатите са същите (горе долу) => моята хипотеза е вярна няма значение от броя нишки (разбира се че не е хипотеза, напълно ми беше ясно че така ще стане от самото начало)

```
rpopov@TrashPiano
-----
OS: Manjaro Linux x86_64
Host: Aspire A515-51G V1.12
Kernel: 5.6.16-1-MANJARO
Uptime: 2 hours, 21 mins
Packages: 1163 (pacman)
Shell: zsh 5.8
Resolution: 1920x1080
DE: Xfce
WM: Xfwm4
WM Theme: Matcha-sea
Theme: Matcha-dark-sea [GTK2], Adwaita [
Icons: Papyrus-Maia [GTK2], Adwaita [GTK
Terminal: alacritty
Terminal Font: monospace
CPU: Intel i5-8250U (8) @ 1.600GHz
GPU: NVIDIA GeForce MX150
GPU: Intel UHD Graphics 620
Memory: 1723MiB / 7837MiB
```



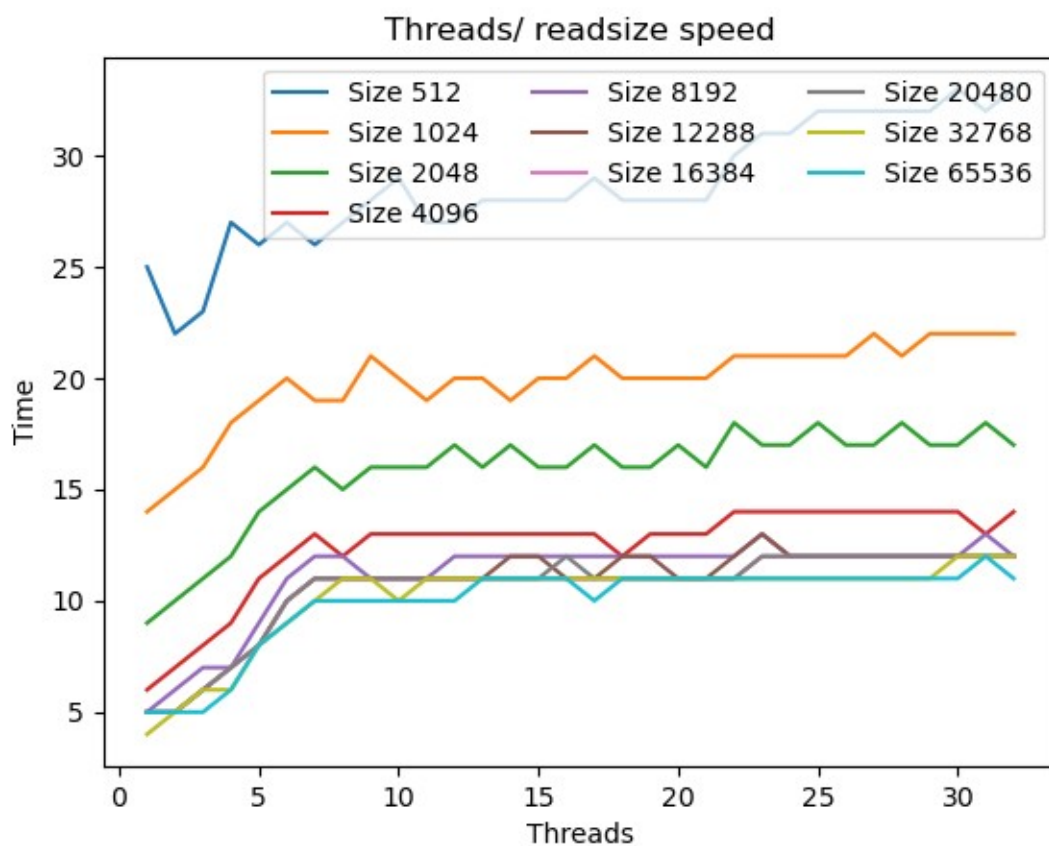
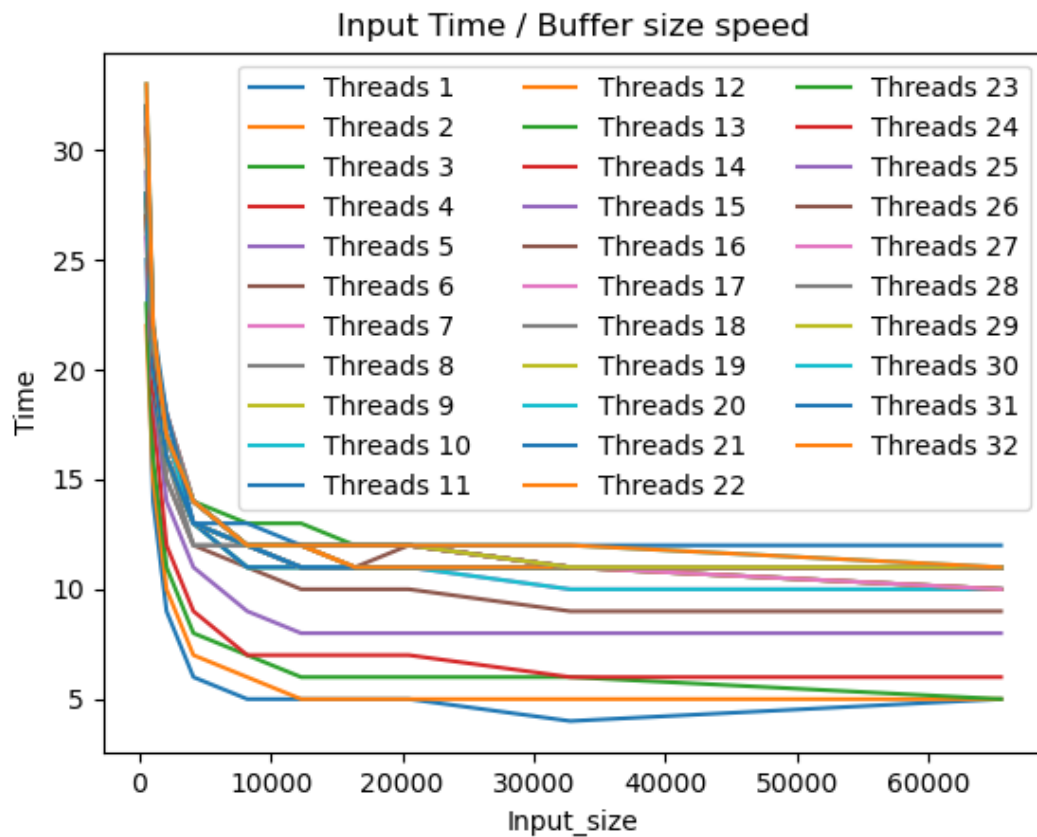
Подход 1: почти същото както предишната графика. Само че малко по голям диапазон





Но резултатите от втората са по важни все пак, и те са същите, отново потвърждава моето твърдение

Подход 2: Отново няма особена разлика между двете



Заклучение:

Четенетп тп файл не е парализируем процес. Повече нишки само го забавят. Но забравих да спомена има и 4-ти вариант за реализация. При него всяка нишка има диапазон от файла който трябва да обработи и си отваря файла и чете в този диапазон без да има нужда от синхронизация. Заклучението е същото като тук с малко изключение. За големи буфери които се четат от един момент нататъка спира да има значение колко нишки има. Скалирането се изжда ясно обаче когато четем 512 байта на всяко четене, разбира се обаче имплементацията не е на C а на golang. Това се намира в github-линк от началото + кратък доклад с графики;

https://github.com/god-rosko-vs-the-bugs/Rsa_Project_Rosen.git