Курсова работа

по Системи за паралелна обработка

Тема: „Алгоритъм ха Хъфман за компресия на данни – честотни таблици“

Изготвен от:

Даниел Делчев, фн:81211, Компютърни Науки,

3 курс, 2 поток, 6 група

Летен семестър 2017г

Проверил: .....

(ас. Христо Христов)

Условие на поставената задача

Условието на поставеният проблем е да се предостави разумно паралелизирана имплементация на алгоритъма на Хъфман за компресия без загуба на данни в частта му отнасяща се до построяване на честотната таблица. С други думи по даден текстов или бинарен файл, трябва да се построи честотна таблица , съдържаща информация колко пъти се среща всеки байт.

Анализ на поставената задача

Тъй като в естеството си задачата не се състои в провеждането на изчисления, а в четене на файл и обработка на прочетените данни, то от гледна точка на гранулярност е напълно логично да изберем най-едрата (всяка нишка ще отговаря за по равен брой байтове от файла). Няма опасност за това някоя нишка да получи „по-тежък вход от данни”, защото няма как да твърдим, че дадена група байтове са по-сложни за прочит и обработка от друга. Също така е най-логично да се използва SPMD Master-Slave модел , при който главната нишка да разпредели задачите на поднишки, които да изчака и след това да обработи резултата от работата им.Балансирането бива статично.След като е ясно каква гранулярност ще използваме, какъв модел на работа между нишките и, че няма опастност от това дадена нишка да се претовари с по-сложен вход от друга изглежда като че ли няма голяма вариация в имплементациите за решение, които бихме могли да изберем, но това е илюзиорно.

Най-значимият проблем,който трябва да се реши преди да се подходи към каквото и да е решение на проблема е как точно ще бъде прочетен, запазен и обработен входният файл, който се предполага, че е поне няколко GB.

Има няколко модели, които са на пръв поглед смислени:

1) Може да се прочете целият файл последователно в паметта, след което да бъде разделен поравно на всяка от нишките, те да преброят байтовете за своите части, след което Master нишката да събере честотните таблици в една.
   Позитиви:
   Проста имплементация, входно изходната работа ще бъде извършена само веднъж.
   Негативи:

Времето за цялостен прочит на файл със размери от няколко GB е значително, на фона на работата която ще извършим над самият файл.

2) Може да оптимизираме горният подход , като реализираме Producer-Consumer подход, като една нижка Producer ще чете части от файла, и когато е прочела достатъчно от него ще го даде на Consumer нишка, която да извърши броенето на байтовете.
   Позитиви:
   Работата по обработка на данните се извършва паралелно с тази по извършване на входно изходните операции.
   Негативи:
   По-сложна имплементация. Без значение, че го правим поетапно, отново прочитаме целият файл последователно в паметта, чрез буфериране. Дори цялата работа по броенето на байтовете да се извърши докато прочетем файла, работата по четенето ще е значително по-бавна.

3) Друг вариант, е да възложим на всяка Slave нишка едновременно да прочита и да брои байтовете от своята част от файла, след което да върне резултата на Master нишката.
   Позитиви:
   Сравнително интуитивна имплементация, модулярност на работата на всяка нишка (всяка изцяло е с грижата да прочете и обработи входа си, независимо от другите).
   Негативи:
   Паралелното четене от диска е изключително вероломно, ако секторите в които четем са значително отдалечени, то главата ще се прехвърля от един сектор в друг далечен доста често, което би забавило значително процеса. Може би , ако секторите не са раздалечени (четем паралелно само от един файл), ОС ще е достатъчно умна да сътвори нещо с КЕШ паметта и да направи смисъл от паралелното четене (не сме сигурни).

4) Възможно е да не прочитаме последователно файлът, а да използваме функцията mmap за да го заредим виртуално в адресното пространство, така random достъпът до части от файла ще е по-оптимален.На всяка Slave нишка ще даден отсек от паметта заделена с mmap.
   Позитиви:

Не е нужно да прочитаме последователно файла чрез буфериране. Така избявгваме гигантски overhead.
Негативи:
По-сложна имплементация. Разчитаме, че ще може да map-нем файл с размеи около 2GB в паметта.

5) Ако имаме ресурси, и условията го позволяват, може да преположим, че файлът, върху който ще работим е разпрострян на няколко дискови устройства, тоест паралелното четене на файла е възможно в истинският смисъл на идеята. Върху такава система приложен модел Producer-Consumer описан в модел 2), би работил страхотно
Позитиви:
Истинско утилизиране на ресурсите.
Негативи:
Доста сложна имплементация.Евентуална работа за preprocess на входния файл.Ограничение върху машините, върху които моделът би работил (не всички машини ще имат повече от 1 диск).

Тъй като модел 5) е извън възможностите ни, по хардуерни съображения, ще се спрем върху модели 4) и 3). Ще представим приложеният алгоритъм по имплементацията им (той е на 95% еднакъв за двата модела), както и статистика от проведените опити. Имплементацията е на езикът за програмиране C++. За провеждане на тестове, тяхното осредняване и визуализиране са изплозвани shell скриптове, както и R скриптове.

Както се изисква по условие при стартиране на изходният файл на програмата се очакват аргументи на командния ред –t или –threads за брой нишки, -q или –quiet за тих режим, -f или -file за указване на име на входен файл, както и параметър –test ,указващ дали да се състави csv файл с резултат от изпълнението.

Имплементация на моделите и изпълнение на поставената задача:

Описание на алгоритъма:

В същността си и двата алгоритъма по изграждане на модел 3) и модел 4) са почти идентитични. Първо ще опишем общото, а в последствие разликата в имплементацията между двата. След parse-ване на аргументите от комания ред, от главната Master нишка се извиква функция, която да изчисли, размера на входният файл, да изчисли каква част от него и в какви интервали трябва да се обработят от Slave нишките, след което създава и извиква за изпълнение Slave нишките и чака тяхното приключване. Slave нишките parse-ват своята част от файла създавайки честотна таблица именно върху тази част.След като и последната Slave нишка завърши, Master нишката събира почленно таблиците от Slave нишките за да образува крайната честотна таблица.Разликата между двата модел е в това как точно се достъпва файла от Slave нишките. При модел 3) Master нишката възлага от къде до къде да работи Slave нишката върху файла, след което тя създава fstream към него със съответния offset.При модел 3 Master нишката предварително извиква примитива mmap върху файла, за да го зареди виртуално в адресното пространство на програмата, след което подава на Slave нишките съответния отсек от тази памет върху, която те да работят.

Описание на по-важните функции и структури:

Функцията **void ParseCommandLineArguments(…)** обработва аргументите от комадния ред и ги записва в структура състояща се единствено от променливи моделиращи тези входни аргументи. Тази структура се използва глобално в останалите функции за достъп до аргументите от комания ред.

Класът, с който сме моделирали честотната таблица е **class FrequencyTable**. Член данните му са единствено статична константна член данна size=256 и масив от uint32 с размер size. Освен тях в класа има и няколко метода и оператори за достъп до елементи на масива, и за събиране поелементно на таблицата с друга таблица.Обръщаме внимание, че сме избрали този подход вместо изплозване на готова структура от рода на ConcurentHashMap за да избегнем нуждата от заключване, тъй като при само 256 байта е доста вероятно в един и същ момент Slave нишките да прочетат еднакъв байт, и да искат да вдигнат брояча за него едновременно, при което би било нужно да се чака за ключалка.

Функцията в която се извършва по-голямата част от работата на Master нишката е **void buildFrequencyTable(…) .**Тя определя размера на файла, разделя го на части по равно за всяка една от n-те нишки (ако размера не е

кратен на n, последната нишка е с по-малка част).В модел 4) файлът се map-ва в паметта, след което се стартират нишките,като се им се подава частта от map-натата памет отредена за тях, изчакват се, след което резултата от тях се събира и се съставя крайната честотна таблица. В модел 3) map-ване не се извършва, а само се подава отсекът, който е отреден на всяка от Slave нишките.

void reader() е функцията, изпълнявана от системно независимите нишки pthread (C++11). В модел 3) те създават ifstream поток към файла ,изместват го със нужният offset за да стигнат своята част от файла, след което четат байт по байт файла във формат unsigned char, който се cast–ва до int между 0 и 255 и в честотна таблица отредена за тази нишка се вдига бройката на срещане на конкретният байт на позицията в масива на таблицата.В модел 4) се случва същото с тази разлика, че не се съставя ifstream а се работи с map-натата памет.

Засичането на времето за изъплнение е извършено с библиотеката chrono.

Тестова постановка и Резултати:

Тестовете са проведени на предоставеният сървър ats24.rmi.yaht.net, разполагащ с 24 ярда и 64GB RAM. За улеснение за build-ване на проекта се иползва makefile. Проекта е компилиран с g++ compiler version 4.8.5 20159623 (Red Hat 4.8.5-11) (GCC) с ниво на оптимизация O3.За провеждане на тестовете е използван скрипт , стартираш програмата върху близо 2GB двоичен файл по 100 пъти, с 1,2,3....32 нишки ,като резултатите се записват в CSV файлове, които са обработени с R скриптове за получаване на осреднените резултати във вид на таблици и графики. Следва тяхното представяне:

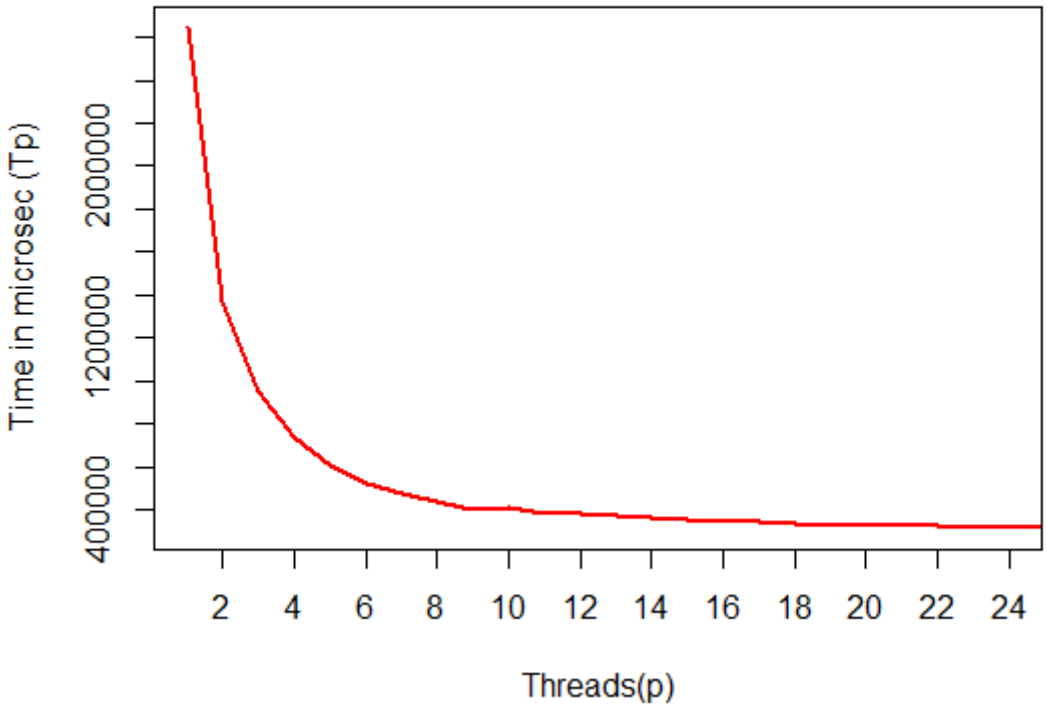
В следващите графики резултатите от модел 4)(mmap) са оцветени винаги в червено, а тези на модел 3)(filestreams) винаги в синьо

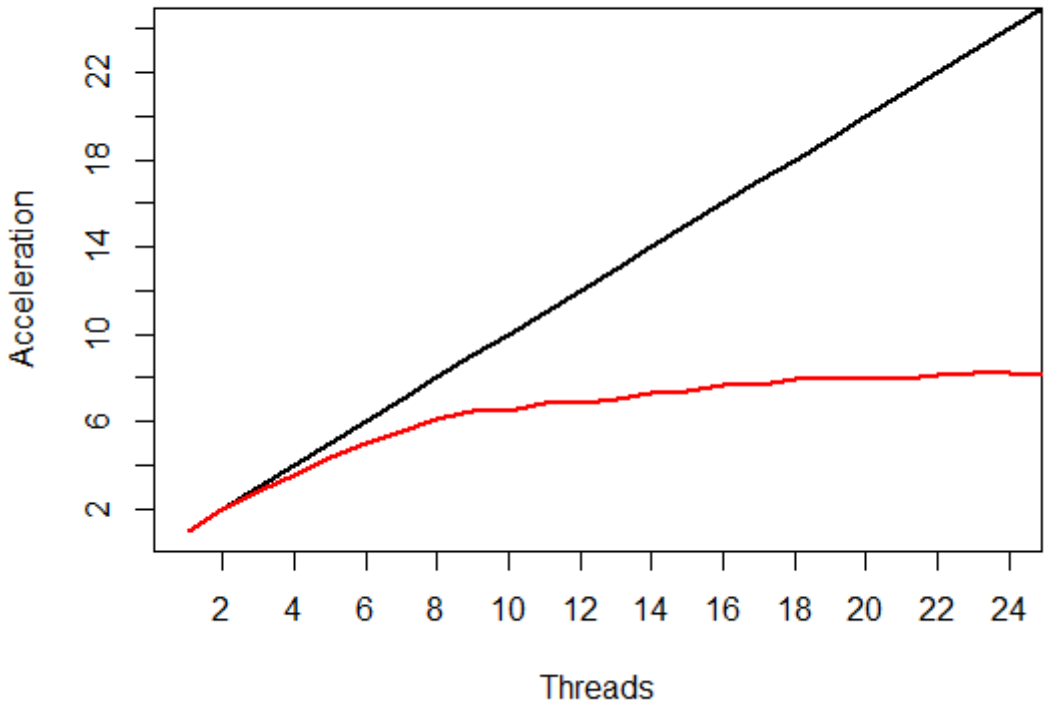
Първо ще представим резултатите от замерванията на модел 4), имплементиращ решение с mmap:

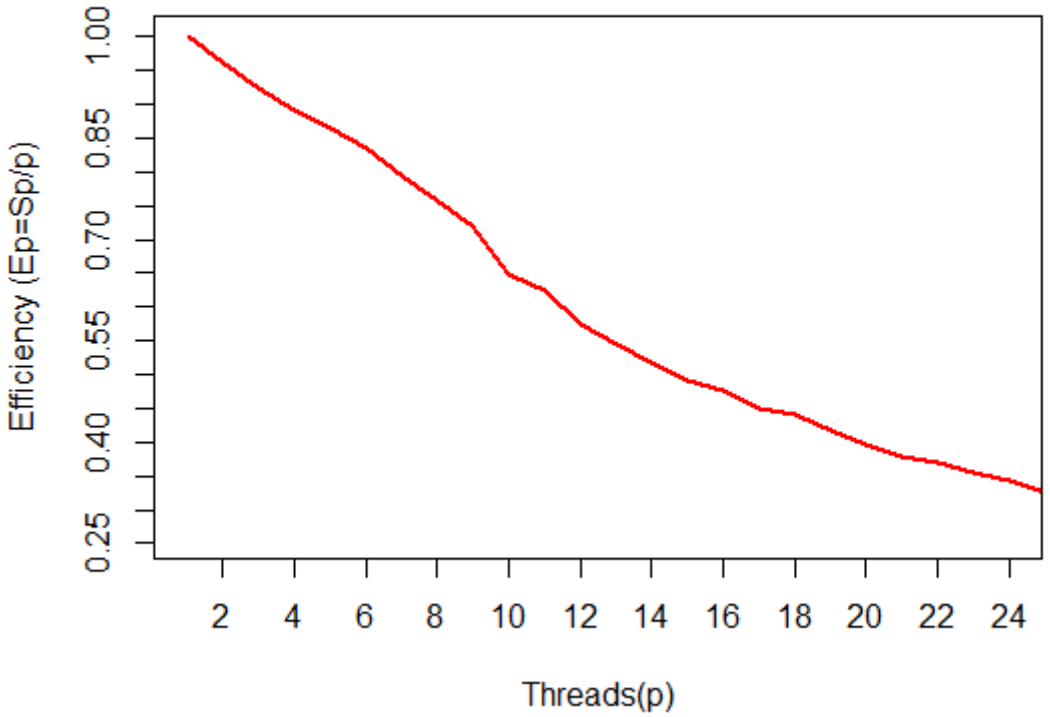
Таблица с резултати Sp = T1 / Tp , Ep = Sp / p , модел(4))

Threads (p)	Time in microsec (Tp)	Acceleration (Sp)	Efficiency (Ep)
1	2644813,31	1	1
2	1373355,13	1,925804369	0,962902185
3	953934,81	2,772530452	0,924176817
4	743169,41	3,558829621	0,889707405
5	611566,56	4,324653248	0,86493065
6	528309,67	5,006180012	0,834363335
7	476270,32	5,553176839	0,793310977
8	435968,85	6,066518996	0,758314874
9	407750,26	6,486355913	0,720706213
10	408441,79	6,475373908	0,647537391
11	384903,08	6,871374763	0,624670433
12	384157,64	6,884708345	0,573725695
13	374360,95	7,064874982	0,543451922
14	364477,06	7,256460283	0,518318592
15	358535,15	7,376719716	0,491781314
16	346083,06	7,64213455	0,477633409
17	344945,42	7,667338531	0,451019914
18	333896,63	7,92105422	0,440058568
19	332628,26	7,951258591	0,418487294
20	333507,9	7,930286839	0,396514342
21	332679,87	7,95002508	0,378572623
22	324735,66	8,144511477	0,370205067
23	322790,92	8,193580259	0,35624262
24	320146,99	8,261246842	0,344218618
25	324750,15	8,144148078	0,325765923
26	318849,35	8,294868125	0,319033389
27	315944,84	8,371123611	0,310041615
28	319497,46	8,278041741	0,295644348
29	318939,82	8,292515215	0,285948801
30	315797,09	8,375040156	0,279168005
31	314314,8	8,41453635	0,271436656
32	319616,22	8,274965864	0,258592683

Следва представяне на графики за времето за изпълнение в микросекунди, графика на ускорението и графика на ефикасността:


Time in microsec (Tp)
400000
1200000
2000000
2
4
6
8
10
12
14
16
18
20
22
24
Threads(p)


Acceleration
2
6
10
14
18
22
2
4
6
8
10
12
14
16
18
20
22
24
Threads



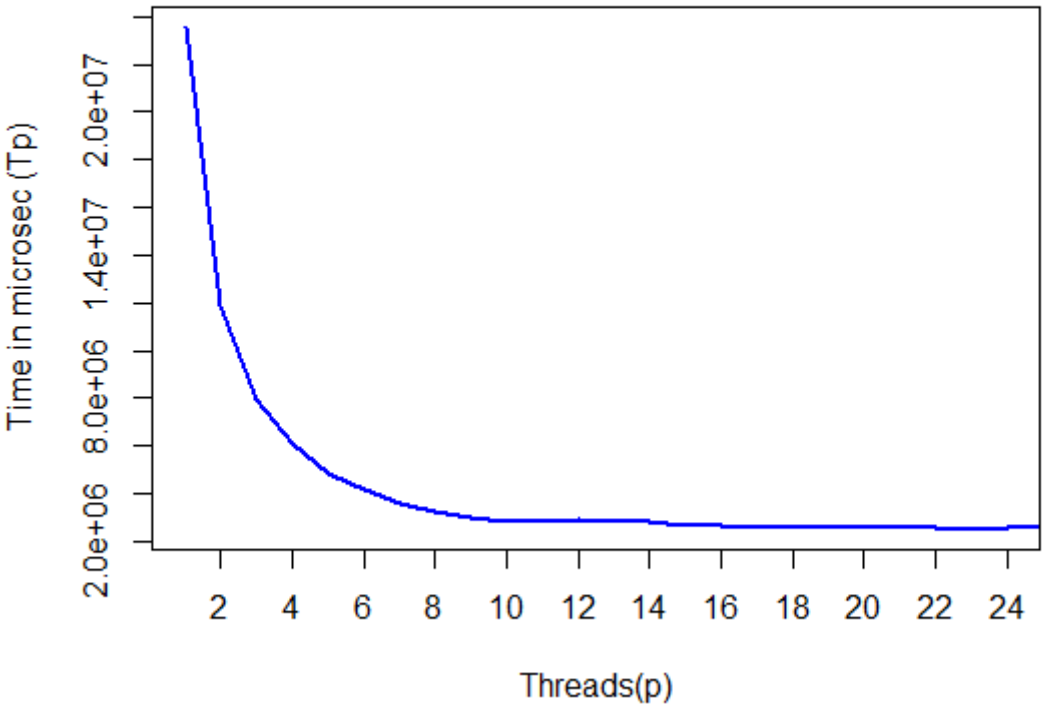
Ускорението не е главозамайващо, около 8.2 пъти при 20-24 нишки, съответно и ефективността е далеч от желаната, но ефекта е на лице при 1 нишка времето за създаване на честотна таблица на 2 GB файл е 2.64 секунди, при 24 нишки времето за това е 0.32 секунди.Модел 4 основаващ се на зареждането на входния файл с mmap не е особено скалируем, но в осезаема степен ускорява изпълнението.

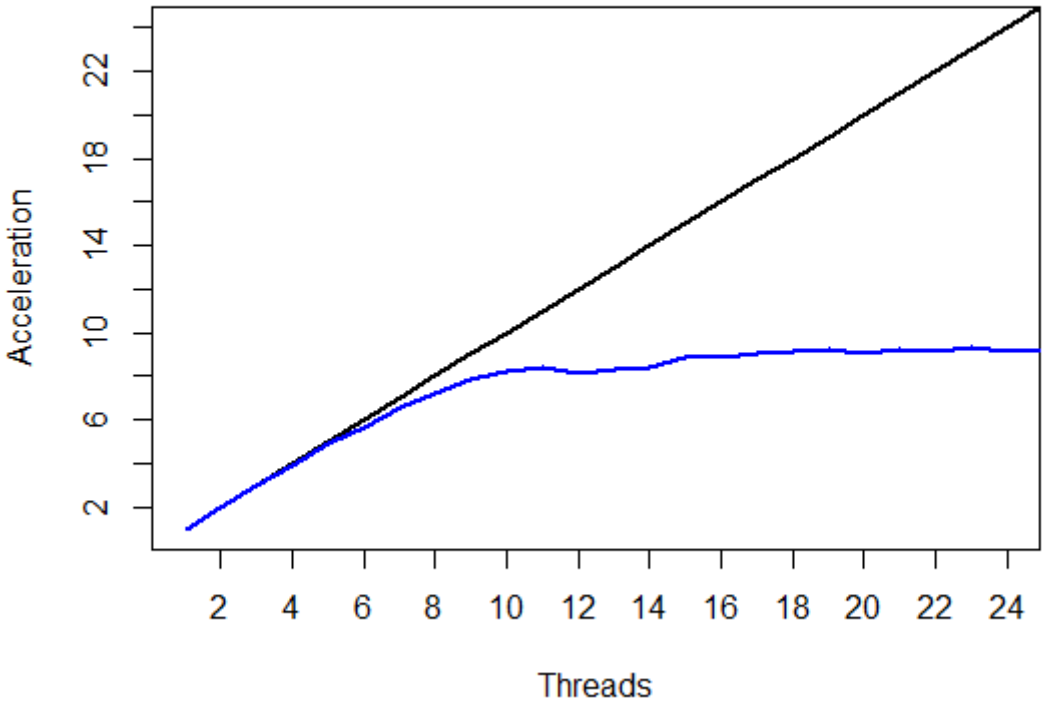
Следва да разгледаме резултатите от модел 3):

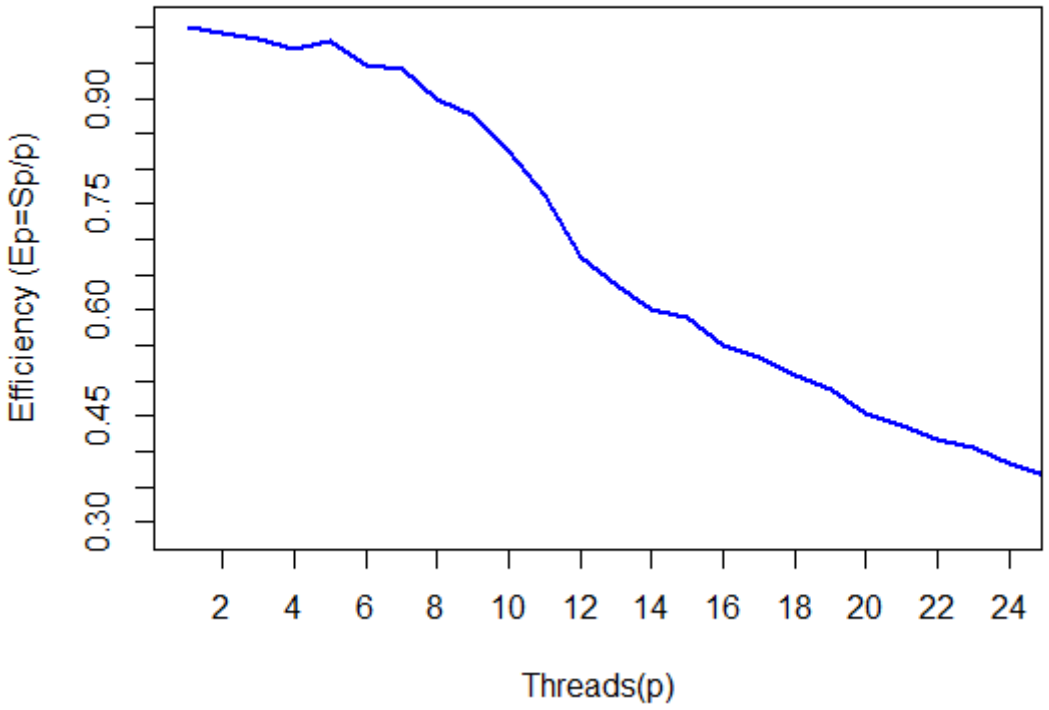
Таблица с резултати Sp = T1 / Tp , Ep = Sp / p , (модел 3))

Threads (p)	Time in microsec (Tp)	Acceleration (Sp)	Efficiency (Ep)
1	23543268,74	1	1
2	11858424,7	1,98536225	0,992681125
3	7983636,71	2,948940388	0,982980129
4	6067679,76	3,880110631	0,970027658
5	4804317,82	4,900439484	0,980087897
6	4149356,61	5,673956459	0,94565941
7	3577487,8	6,580950113	0,94013573
8	3279499,12	7,178922109	0,897365264
9	2982770,38	7,893087882	0,877009765
10	2850540,27	8,25923036	0,825923036
11	2810473,9	8,376974695	0,761543154
12	2906307,46	8,100749513	0,675062459
13	2847878,16	8,266950838	0,635919295
14	2791812,72	8,432968505	0,602354893
15	2655413,96	8,8661388	0,59107592
16	2670511,67	8,816014176	0,551000886
17	2598026,75	9,061980882	0,533057699
18	2574214,74	9,145806049	0,508100336
19	2542349,34	9,260438119	0,48739148
20	2593250,94	9,078669703	0,453933485
21	2562896,91	9,186194204	0,437437819
22	2567484,36	9,169780781	0,416808217
23	2527760,7	9,313883525	0,404951458
24	2567051,42	9,171327289	0,382138637
25	2570714,84	9,158257607	0,366330304
26	2546979,14	9,243604853	0,355523264
27	2605762,41	9,035078812	0,334632549
28	2573679,16	9,147709282	0,326703903
29	2559260,27	9,199247539	0,317215432
30	2539141,22	9,272138373	0,309071279
31	2541419,04	9,263827952	0,29883316
32	2532097,86	9,297930033	0,290560314

Следва представяне на графики за времето за изпълнение в микросекунди,графика на ускорението и графика на ефикасността:


Time in microsec (Tp)
2.0e+06
8.0e+06
1.4e+07
2.0e+07
2
4
6
8
10
12
14
16
18
20
22
24
Threads(p)

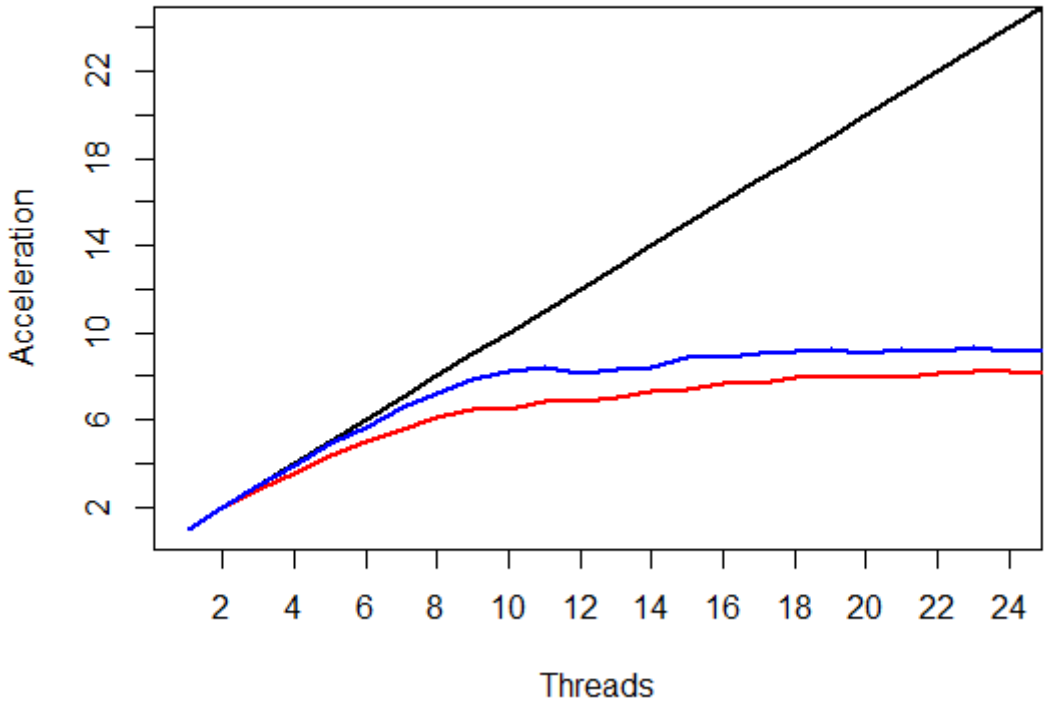

Acceleration
2
6
10
14
18
22
2
4
6
8
10
12
14
16
18
20
22
24
Threads

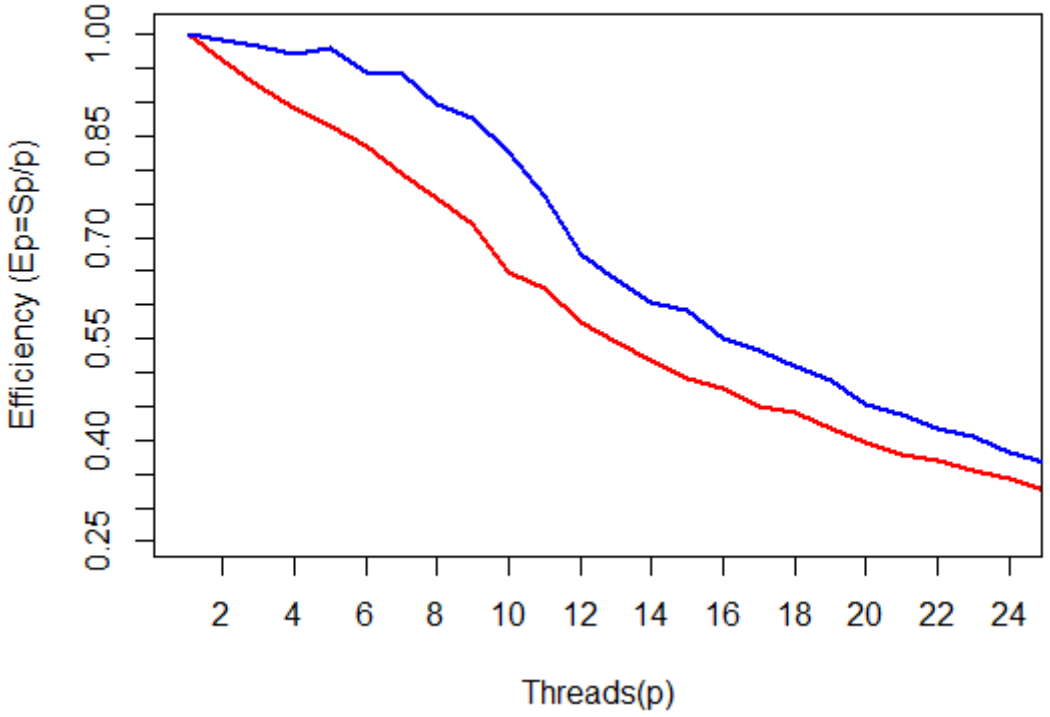


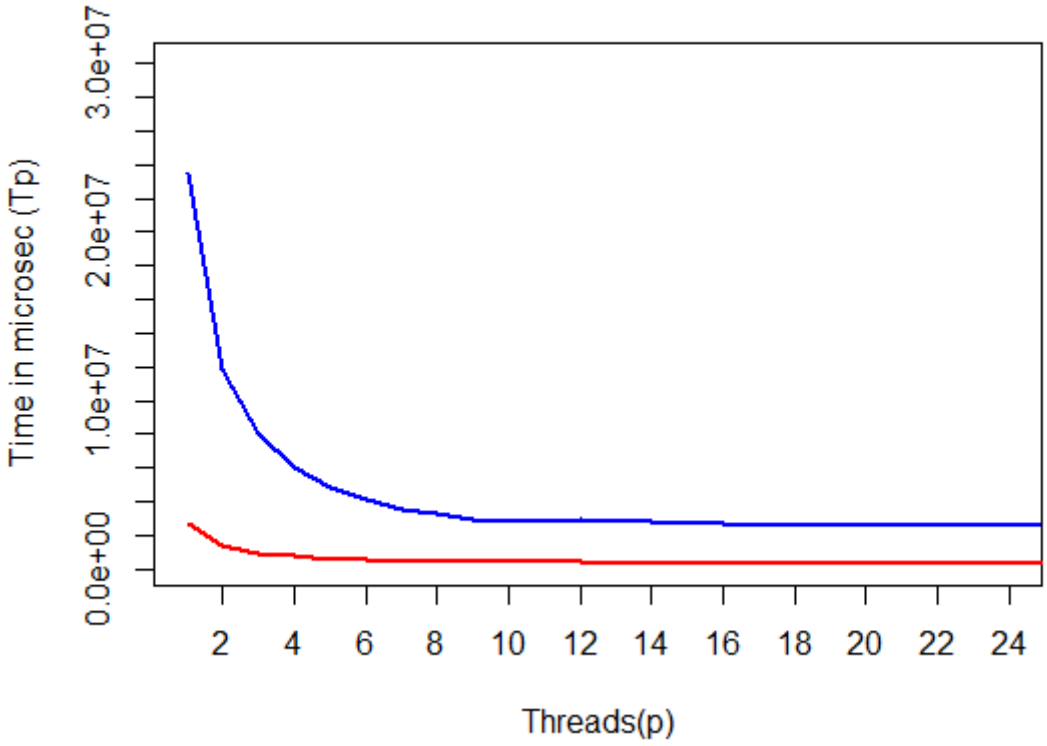
При обработката на същият входен файл ускорението е по-добро, достига 9.3 пъти при 23 нишки, съответно и ефективността е по-добра от тази при модел 4) .На пръв поглед модел 3) е по скалируем, достига по-голямо ускорение и изглежда за предпочитане пред модел 3, това обаче е измамно. Наслагваме графиките от двата модела:

ЧЕРВЕНО – МОДЕЛ 4) (MMAP)

СИНЬО – МОЕДЛ 3) (ifstreams)


Acceleration
2
6
10
14
18
22
2
4
6
8
10
12
14
16
18
20
22
24
Threads


Efficiency (Ep=Sp/p)
0.25
0.40
0.55
0.70
0.85
1.00
2
4
6
8
10
12
14
16
18
20
22
24
Threads(p)



Модел 3) в действителност ускорява по-бързо според броя на нишките, също е и по-ефикасен, но е значително по – бавен от модел 4), което е видно от послената приложена графика. За обработката на един и същ файл при 1 нишка на модел 3) са нужни 23.5 секунди, а на модел 4) 2.64. При 24 нишки на модел 3) са нужни 2.5 секунди, а на модел 4) 0.32. Оказва се , че въпреки ,че модел 3 има по – добра графика на ефективност и ускорение ,модел 4) е много по-добър тъй като при 1 нишка е почти толкова бърз колкото модел 3) при 24 нишки.

В крайна сметка, както очаквахме модел 4) от петте модела разгледани в анализа на задачата е най-добър избор (изключвайки модел 5) , който нямахаме възможност да имплементираме). Модел 3) , който също беше имплементиран и истестван е по лош от модел 4), а модели 1) и 2) също основавайки се на последователен прочит на файла не биха могли да са по добри от модел 3) и съответно от 4). Интересно би било да се разгледат имплементация и замервания на модел 5) , базиран на четене на входният файл от няколко дискови устройства едновременно, но нямаме възможност за такъв опит.