Обхождане на граф в дълбочина

Проект по Системи за паралелна обработка

14 юни 2019 г.

Изготвил: Никола Стоянов
ф.н. 81151, КН, минали години

Проверил: .....
/ас. Христо Христов/

1 Цел на проекта

Задачата е да се реализира паралелен алгоритъм за обхождане на граф в дълбочина (Depth First Search). Графът може да бъде ориентиран или неориентиран, свързан или слабо свързан. Целта е да се постои дърво на обхождането.

Програмата трябва да отговаря на следните изисквания:

- позволява въвеждане на граф от входен файл или генериране на произволен граф по подадени брой върхове и ребра
- позволява извеждане на резултата от обхождането в избран изходен файл
- позволява задаване на максималния брой нишки, които се използват за решаване на задачата

2 Реализация

Проектът е реализиран като приложение за командния ред. Поддържа се следния интерфейс

```
./parallel_dfs gen -n 20 -m 40 -t 4 --algo par_mat
```

- `gen` - режим на работа. Единственият имплементиран е gen - генерирай случаен граф и направи обхождане по него.
- `-n 20` - брой върхове
- `-m 40` - брой ребра
- `-t 4` - колко нишки да използва
- `--algo par_mat` - кои алгоритми и стуктури от данни да използва. Вариантите са `seq_list`, `par_list`, `cheat_list`, `seq_mat`, `par_mat`, `cheat_mat`

Всички аргументи могат да се видят с `./parallel_dfs --help`

Следните изисквания към проекта не са имплементирани:

- въвеждане на граф от входен файл
- извеждане на резултата в изходен файл (има опция за извеждане на резултата на стандартния изход в "debug"формат).

Използван е езикът Rust. Използвани са няколко външни библиотеки за различни детайли, но по-основно е използвана библиотеката rayon. Rayon е библиотека която цели да предостави удобен интерфейс за паралелна обработка на данни. (*Забележка* - една от причините да използвам rayon е че покрай този проект исках да разгледам какви възможности дава библиотеката. Другата е, че rayon имаше най-бързата имплементация на thread pool, която успях да намеря).

Rayon работи с глобален thread pool към който се добавят задачи. В този проект са използвани главно два начина за създаване на задачи:

- `rayon::scope()` - създава "блок от който могат да се пускат задачи в глобалния thread pool със `scope.spawn(fn)`. След края на блока се блокира докато всички пуснати от него задачи не приключат.
- паралелни итератори - `collection.par_iter().for_each(fn)`. Извиква fn върху всеки елемент от коленцията паралелно. Вътрешно се опитва да раздели колекцията на няколко части, така че всяка нишка да работи върху отделна част. Отново самото изпълнение се случва в thread pool-а.

3 Генериране на графа

Решението на задачата може да се раздели на две независими части - генерирането на графа и обхождането му. Реализирани са два варианта за записване на графа - като списъци на съседство и като матрица на съседство.

3.1 Списъци на съседство

Списъците на съседство са в общия случай по-бързи за генериране и обработка и заемат по-малко памет от матрицата, но не позволяват много добро разпаряляване при генерирането на неориентиран граф.

Ориентиран граф `graph::AdjLists::gen_directed()`. За генерирането на ориентиран граф се използва следната стратегия. Графа се разделя на множество части, всяка с фиксиран размер от k върха (k = 128) и $m * \frac{k}{n}$ ребра. За всяка част се пуска задача в thread pool-а, като часта с върхове *a..b* генерира ребра започващи от нея, т.е. ребра *(c, v)*, $a <= c <= b$. Така всяка нишка работи върху отделна част от графа.

Този подход не генерира напълно случаен граф, защото всяка част ще има един и същ брой ребра. Една причина да се генерира на части, а не всеки връх по отделно (т.е. k=1) е за да се намали този ефект.

Има и втори вариант на алгоритъма - `AdjLists::gen_directed_on_threads()`, където графа се разделя на t на брой равни части - по една за всяка нишка. Това служи за сравнение между ръчно направено разпределяне и автоматичното разпределяне от библиотеката rayon. Има micro benchmark за това в директория benches/, разликата е, че rayon е константно с около 10% по-бавен. Всички по-нататъшни резултати използват `AdjLists::gen_directed()`.

Неориентиран граф `graph::AdjLists::gen_undirected()`. Генерирането на неориентиран граф е по-сложна задача, защото за всяко добавено ребро *(u, v)*, трябва да се добави и обратното ребро *(v, u)*. Първата стъпка е да се генерират ребрата *(u, v)* така, че $u > v$. Използва се същия подход като при ориентирания граф, с разликата, че броят ребра, които се генерират за часта с върхове от a до b е $\frac{m*(b^2-a^2)}{n^2}$.

След това трябва да се добавят огледалните ребра. Идеята е да обходим вече генерираните ребра, и за всяко ребро *(a, b)* да добавим реброто *(b, a)*. Ако се опитаме да направим това паралелно трябва да се добави синхронизация, защото може да се срещнат две ребра *(a, b)* и *(c, b)* едновременно, при което се получава едновременно писане в вектора b.

Изпробвани са няколко подхода, но най-бързо се оказа това да се направи последователно на една нишка, поне за размерите на графа с който е тествано.

3.2 Матрица на съседство

Матрицата на съседство е имплементирана чрез конкурентен битов вектор. За целта съм модифицирал структурата предоставяна от една външна библиотека, като съм опростил интерфейса и съм заменил операциите за четене и писане на битове с атомарни такива.

При матрицата разпаряляването на алгоритъма е лесно и в двата случая.

Ориентиран граф `graph::AdjMatrix::gen_directed()`. Разделяме задачата на подзадачи, като всяка подзадача генерира фиксиран брой ребра. Подзадачите се изпълняват едновременно от thread pool. Всяка подзадача генерира случайно ребро *(u, v)* и го записва в матрицата и проверява дали вече съществува с една атомарна операция. Това продължава докато не се генерират 128 ребра.

Неориентиран граф `graph::AdjMatrix::gen_undirected()` Идеята е същата като при ориентирания, с изключението че се генерират само ребра за които $u < v$. Нишката пробва да добави реброто *(u, v)*. Ако това успее, то реброто *(v, u)* също не съществува и никой друг няма да се опита да го добави, затова нишката го добавя и него.

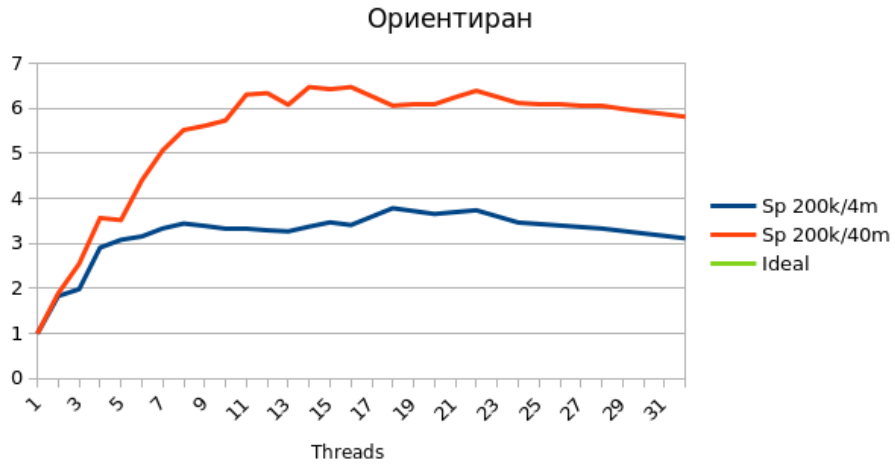
3.3 Резултати

Резултати от измерване върху t5600. Алгоритмите са измерени с два различни входа - 200_000 върха, 4_000_000 ребра и 200_000 върха, 40_000_000 ребра.

В случая на матрица на съседство отчетеното време не включва времето за заделяне на матрицата, което е 2.5 - 3 секунди.

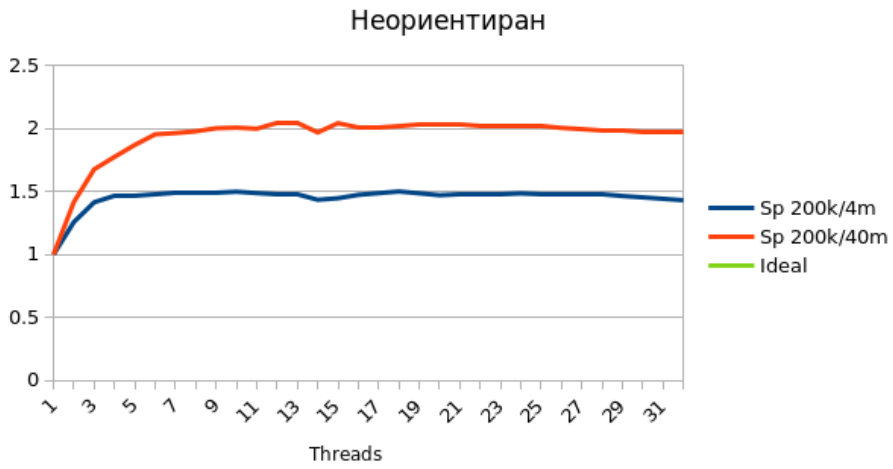
Списъци на съседство - ориентиран граф

нишки	време (200k/4m)	S_p	E_p	време (200k/40m)	S_p	E_p
1	0.389 сек	1	1	5.106 сек	1	1
2	0.212 сек	1.83	0.91	2.696 сек	1.89	0.94
3	0.196 сек	1.98	0.66	2.002 сек	2.55	0.85
4	0.134 сек	2.90	0.72	1.431 сек	3.56	0.89
6	0.123 сек	3.15	0.52	1.159 сек	4.40	0.73
8	0.113 сек	3.44	0.43	0.925 сек	5.52	0.69
10	0.116 сек	3.33	0.33	0.891 сек	5.73	0.57
12	0.118 сек	3.29	0.27	0.805 сек	6.33	0.52
14	0.115 сек	3.37	0.24	0.788 сек	6.47	0.46
16	0.114 сек	3.41	0.21	0.788 сек	6.47	0.40
20	0.106 сек	3.65	0.18	0.836 сек	6.10	0.30
24	0.112 сек	3.46	0.14	0.834 сек	6.11	0.25
28	0.116 сек	3.33	0.11	0.844 сек	6.04	0.21
32	0.125 сек	3.11	0.09	0.878 сек	5.81	0.18



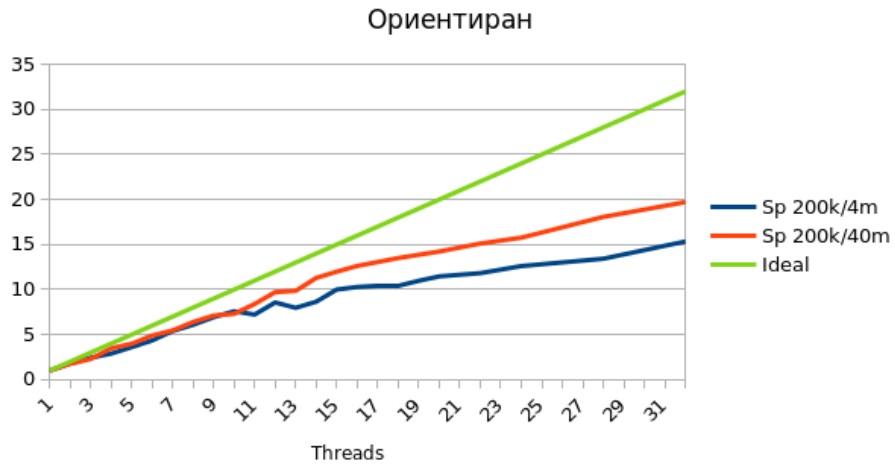
Списъци на съседство - неориентиран граф

нишки	време (200k/4m)	S_p	E_p	време (200k/40m)	S_p	E_p
1	0.729 сек	1	1	8.868 сек	1	1
2	0.580 сек	1.25	0.62	6.262 сек	1.41	0.70
3	0.516 сек	1.41	0.47	5.290 сек	1.67	0.55
4	0.498 сек	1.46	0.36	4.998 сек	1.77	0.44
6	0.493 сек	1.47	0.24	4.541 сек	1.95	0.32
8	0.491 сек	1.48	0.18	4.487 сек	1.97	0.24
10	0.487 сек	1.49	0.14	4.421 сек	2.00	0.20
12	0.494 сек	1.47	0.12	4.350 сек	2.03	0.16
14	0.508 сек	1.43	0.10	4.506 сек	1.96	0.14
16	0.495 сек	1.47	0.09	4.416 сек	2.00	0.12
20	0.496 сек	1.47	0.07	4.362 сек	2.03	0.10
24	0.491 сек	1.48	0.06	4.383 сек	2.02	0.08
28	0.494 сек	1.47	0.05	4.468 сек	1.98	0.07
32	0.510 сек	1.43	0.04	4.503 сек	1.96	0.06



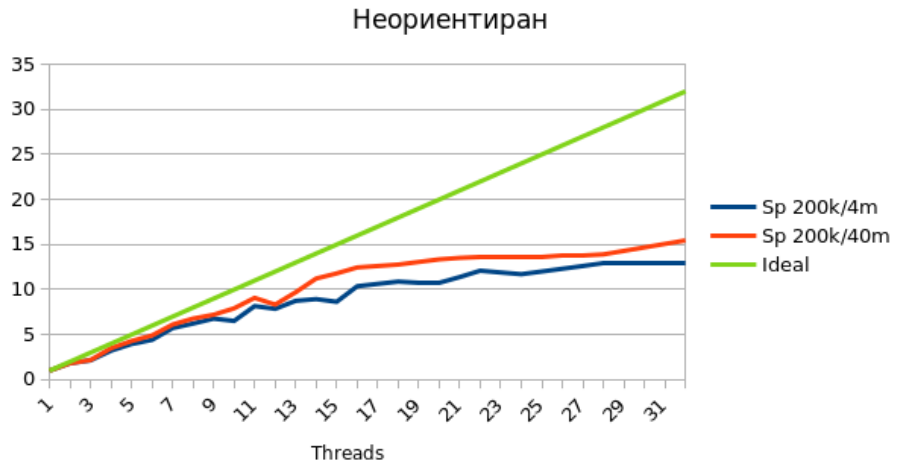
Матрица на съседство - ориентиран граф

нишки	време (200k/4m)	S_p	E_p	време (200k/40m)	S_p	E_p
1	0.717 сек	1	1	7.144 сек	1	1
2	0.398 сек	1.80	0.90	4.065 сек	1.75	0.87
3	0.294 сек	2.43	0.81	3.119 сек	2.29	0.76
4	0.248 сек	2.88	0.72	2.049 сек	3.48	0.87
6	0.165 сек	4.34	0.72	1.459 сек	4.89	0.81
8	0.117 сек	6.08	0.76	1.115 сек	6.40	0.80
10	0.094 сек	7.59	0.75	0.977 сек	7.31	0.73
12	0.083 сек	8.56	0.71	0.735 сек	9.71	0.80
14	0.082 сек	8.65	0.61	0.632 сек	11.28	0.80
16	0.069 сек	10.28	0.64	0.566 сек	12.61	0.78
20	0.062 сек	11.46	0.57	0.502 сек	14.22	0.71
24	0.056 сек	12.60	0.52	0.453 сек	15.75	0.65
28	0.053 сек	13.42	0.47	0.395 сек	18.06	0.64
32	0.046 сек	15.34	0.47	0.362 сек	19.71	0.61



Матрица на съседство - неориентиран граф

нишки	време (200k/4m)	S_p	E_p	време (200k/40m)	S_p	E_p
1	0.943 сек	1	1	9.443 сек	1	1
2	0.514 сек	1.83	0.91	5.196 сек	1.81	0.90
3	0.439 сек	2.14	0.71	4.326 сек	2.18	0.72
4	0.292 сек	3.22	0.80	2.698 сек	3.49	0.87
6	0.213 сек	4.41	0.73	1.920 сек	4.91	0.81
8	0.151 сек	6.23	0.77	1.392 сек	6.77	0.84
10	0.144 сек	6.52	0.65	1.192 сек	7.91	0.79
12	0.119 сек	7.86	0.65	1.133 сек	8.33	0.69
14	0.105 сек	8.93	0.63	0.841 сек	11.22	0.80
16	0.090 сек	10.37	0.64	0.758 сек	12.45	0.77
20	0.088 сек	10.68	0.53	0.707 сек	13.35	0.66
24	0.080 сек	11.70	0.48	0.694 сек	13.60	0.56
28	0.072 сек	12.92	0.46	0.679 сек	13.90	0.49
32	0.072 сек	12.97	0.40	0.610 сек	15.45	0.48



4 Обхождане на графа

4.1 Алгоритми

Резултатът от обхождането на графа е гора от дървета, като всяко дърво е представено като корен и списък от наредени двойки *(родител, дете)*. Списъкът е сортиран в реда в който DFS алгоритъмът е намерил съответното дете. Може да има повече от едно дърво ако графът не е свързан. Имплементирани са три различни DFS алгоритъма.

Seq Стандартен еднонишков dfs алгоритъм. Използва се за база за сравнение.

Cheat За всеки връх се създава задача, която пуска dfs от този връх, и задачите се добавят в thread pool-а. Това значи че в началото ще започнат t на брой задачи - но една за всяка нишка. Когато задача приключи ще се извика следващата на нейно място (която най-вероятно ще види че върха от който почва вече е обходен и ще приключи).

Използва се споделен масив за проверка на това кои върхове са обходени. Първата нишка, която намери връх го маркира за обходен. Ако нишка види вече обходен връх тя го игнорира.

Резултатът от алгоритъма се различава от този на стандартния еднонишков варант, защото графът се разделя изкуствено на няколко части. За сметка на това се печели скорост, защото синхронизацията е минимална и отделните нишки не се застъпват в работата, която вършат.

Par Пуска се еднонишков dfs алгоритъм, който изпълнява само "спускането". Ако в момента сме във връх u то се преминава към първото необходено дете на u. Ако u няма необходени деца, алгоритъма спира. Резултатът от това е стек с всички намерени по време на спускането върхове. Следващата стъпка е връщането назад (backtracking) - при което трябва да се проверят и останалите необходени деца (след първото) на елементите в стека. Тази стъпка се изпълнява паралелно. За всеки елемент от стека се създава подзадача, която изпълнява нормален (пълен) dfs започвайки от този връх. Задачите се добавят в thread pool.

Така пуснатите задачи все още могат да намерят едни и същи върхове. Затова всеки пуснат dfs има цифров приоритет, като dfs-ите с елементи от началото на стека имат по-голям приоритет. Пази се споделен масив, в който за всеки намерен връх се записва приоритета, с който е намерен. Когато една нишка намери връх, тя проверява в масива с приоритетите. Ако този елемент вече съществува с по-голям приоритет, той се смята за обходен и се игнорира. Ако не съществува или има по-малък приоритет, то приоритета се презаписва с по-големия и обхождането продължава.

След приключване на втората стъпка имаме отделни дървета от всяка под-задача. Последната стъпка включва премахване на елементи, които са били обходени повече от веднъж от всички дървета освен това с най-голям приоритет и обединяването на всички дървета в едно. Времето за тези операции е незначително спрямо по-горните стъпки.

Ако след приключването на целият алгоритъм от един връх все още има необходени върхове в графа, алгоритъма се повтаря за тях. Това се случва последователно. Тука не се цели паралелизъм, защото за случаен граф е много вероятно целият или почти целият граф да е една свързана компонента.

Този алгоритъм връща същия резултат като еднонишков dfs. Идеята му е подзадачите да се пуснат в реда в който минимално ще припокриват

работата си, което в случая значи задачите с по-голям приоритет да работят преди задачите с по-малък.

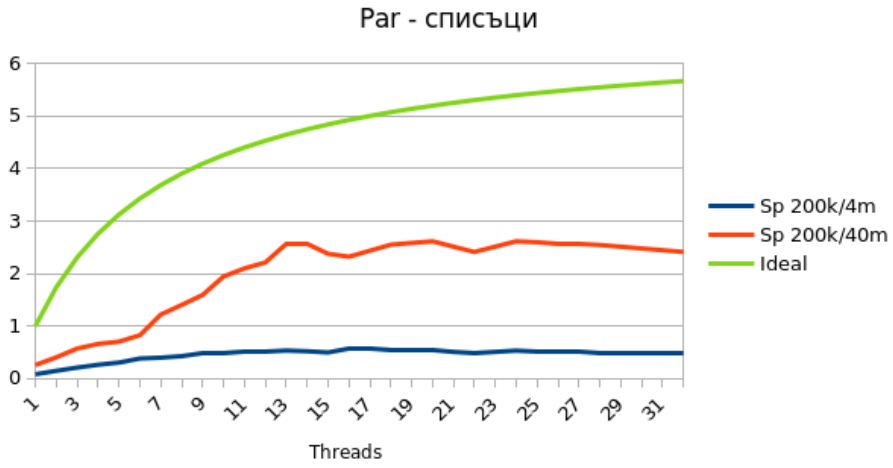
4.2 Резултати

Отново алгоритмите са тествани с два различни входа - генериран ориентиран граф с 200_000 върха, 4_000_000 ребра и 200_000 върха, 40_000_000 ребра.

Понеже `Par` алгоритъма има последователна стъпка максималното достижимо ускорение не е линейно. По измервания отношението (*време за "спускане"/ време за изпълнение на seq*) ≈ 0.15. Затова за идеална ускорение е използвана графиката на $1/(0.15 + \frac{0.85}{t})$.

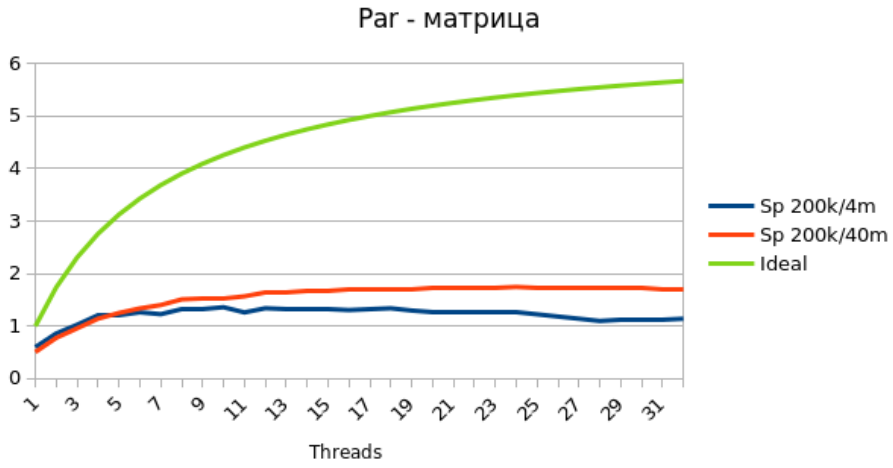
par - списъци на съседство

нишки	време (200k/4m)	S_p	E_p	време (200k/40m)	S_p	E_p
seq	0.096 сек	-	-	0.624 сек	-	-
1	1.141 сек	0.08	0.08	2.398 сек	0.26	0.26
2	0.642 сек	0.14	0.07	1.535 сек	0.40	0.20
3	0.452 сек	0.21	0.07	1.087 сек	0.57	0.19
4	0.362 сек	0.26	0.06	0.942 сек	0.66	0.16
6	0.250 сек	0.38	0.06	0.754 сек	0.82	0.13
8	0.224 сек	0.42	0.05	0.444 сек	1.40	0.17
10	0.198 сек	0.48	0.04	0.320 сек	1.94	0.19
12	0.190 сек	0.50	0.04	0.282 сек	2.21	0.18
14	0.183 сек	0.52	0.03	0.244 сек	2.55	0.18
16	0.165 сек	0.57	0.03	0.268 сек	2.32	0.14
20	0.179 сек	0.53	0.02	0.238 сек	2.61	0.13
24	0.179 сек	0.53	0.02	0.238 сек	2.61	0.10
28	0.194 сек	0.49	0.01	0.245 сек	2.54	0.09
32	0.197 сек	0.48	0.01	0.258 сек	2.41	0.07



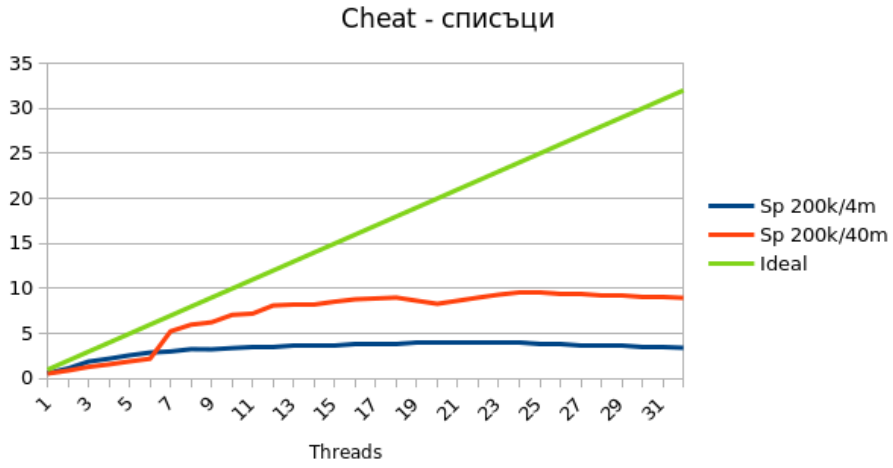
par - матрица на съседство

нишки	време (200k/4m)	S_p	E_p	време (200k/40m)	S_p	E_p
seq	2.175 сек	-	-	3.072 сек	-	-
1	3.621 сек	0.60	0.60	6.054 сек	0.50	0.50
2	2.525 сек	0.86	0.43	3.954 сек	0.77	0.38
3	2.118 сек	1.02	0.34	3.202 сек	0.95	0.31
4	1.805 сек	1.20	0.30	2.689 сек	1.14	0.28
6	1.722 сек	1.26	0.21	2.296 сек	1.33	0.22
8	1.627 сек	1.33	0.16	2.035 сек	1.50	0.18
10	1.599 сек	1.36	0.13	2.030 сек	1.51	0.15
12	1.622 сек	1.34	0.11	1.881 сек	1.63	0.13
14	1.639 сек	1.32	0.09	1.835 сек	1.67	0.11
16	1.666 сек	1.30	0.08	1.807 сек	1.70	0.10
20	1.734 сек	1.25	0.06	1.782 сек	1.72	0.08
24	1.716 сек	1.26	0.05	1.759 сек	1.74	0.07
28	1.979 сек	1.09	0.03	1.777 сек	1.72	0.06
32	1.905 сек	1.14	0.03	1.805 сек	1.70	0.05



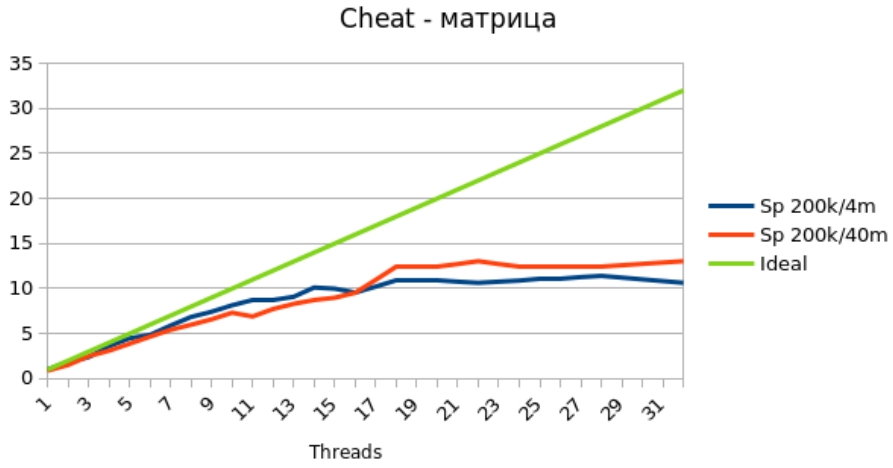
cheat - списъци на съседство

нишки	време (200k/4m)	S_p	E_p	време (200k/40m)	S_p	E_p
seq	0.099 сек	-	-	0.584 сек	-	-
1	0.165 сек	0.59	0.59	1.029 сек	0.56	0.56
2	0.086 сек	1.14	0.57	0.650 сек	0.89	0.44
3	0.052 сек	1.89	0.63	0.448 сек	1.30	0.43
4	0.044 сек	2.21	0.55	0.369 сек	1.58	0.39
6	0.034 сек	2.90	0.48	0.264 сек	2.21	0.36
8	0.030 сек	3.27	0.40	0.097 сек	5.99	0.74
10	0.029 сек	3.39	0.33	0.082 сек	7.07	0.70
12	0.027 сек	3.57	0.29	0.072 сек	8.12	0.67
14	0.027 сек	3.66	0.26	0.070 сек	8.28	0.59
16	0.026 сек	3.77	0.23	0.066 сек	8.80	0.55
20	0.024 сек	4.02	0.20	0.070 сек	8.30	0.41
24	0.024 сек	3.99	0.16	0.060 сек	9.65	0.40
28	0.027 сек	3.67	0.13	0.063 сек	9.25	0.33
32	0.028 сек	3.42	0.10	0.065 сек	8.96	0.28



cheat - матрица на съседство

нишки	време (200k/4m)	S_p	E_p	време (200k/40m)	S_p	E_p
seq	2.264 сек	-	-	3.131 сек	-	-
1	2.237 сек	1.01	1.011	3.515 сек	0.89	0.89
2	1.220 сек	1.85	0.927	2.063 сек	1.51	0.75
3	0.950 сек	2.38	0.794	1.257 сек	2.49	0.83
4	0.627 сек	3.60	0.901	1.009 сек	3.10	0.77
6	0.465 сек	4.86	0.810	0.670 сек	4.66	0.77
8	0.330 сек	6.85	0.856	0.523 сек	5.98	0.74
10	0.277 сек	8.15	0.815	0.427 сек	7.32	0.73
12	0.258 сек	8.75	0.729	0.404 сек	7.73	0.64
14	0.223 сек	10.1	0.722	0.358 сек	8.73	0.62
16	0.236 сек	9.55	0.597	0.328 сек	9.53	0.59
20	0.207 сек	10.8	0.544	0.252 сек	12.4	0.62
24	0.207 сек	10.8	0.453	0.252 сек	12.4	0.51
28	0.198 сек	11.4	0.407	0.251 сек	12.4	0.44
32	0.212 сек	10.6	0.332	0.240 сек	13.0	0.40


Cheat - матрица
35
30
25
20
15
10
5
0
1
3
5
7
9
11
13
15
17
19
21
23
25
27
29
31
Threads
Sp 200k/4m
Sp 200k/40m
Ideal