

traytor - разпределен рейтрейсър

Диана Генева <dageneva@gmail.com> ФН 81008

Ангел Ангелов <hextwoa@gmail.com> ФН 81050

2016

<https://github.com/DexterLB/traytor>

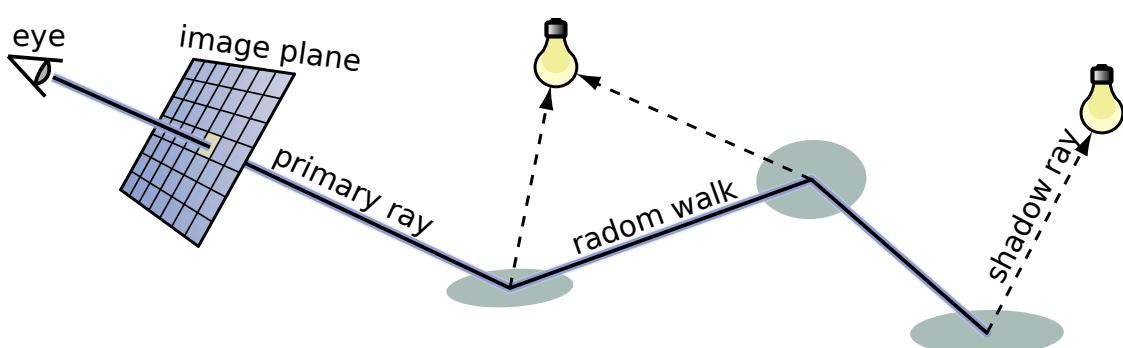


1 Описание

Проектът ни имплементира рейтрейсър, който по дадена сцена генерира картина. Симулира се светлината, както би се държала в реалната (Нютонова) вселена, като се изстрелят лъчи от камерата, и се симулира как те се удрят в обекти из сцената, докато стигнат до повърхност, излъчваща светлина. Така получаваме пътя, изминат от един фотон до камерата - и нещата, които се случват с този фотон, могат на практика да са само две:

- Отражение
- Пречупване

Като при всяко едно такова събитие лъчът може да си промени цвета по някакъв начин. Това е алгоритъмът **path tracing** - той е много прост, но също така идеално възпроизвежда физиката: всичко, което можем да видим с очи, може да се представи в сцена. Проблемът на този алгоритъм е, че е **много, много, много бавен**. Макар че можем да получим всичко само с тези два материала, се налага да имаме например "грапаво отражение" (Ламбъртов материал), за да симулираме твърди повърхности - иначе трябваше да моделираме всяка една грапавина върху повърхността.



2 Архитектура

Рейтрайсърът е реализиран на Go. Не са използвани други външни библиотеки освен стандартната.

Потребителят има досег до следните части от проекта:

- traytor render - команда за локално рендиране на сцена
- traytor worker - работник, който се стартира на няколко машини
- traytor client - клиент, който рендира разпределено на няколко работници
- Blender exporter - скрипт на Python, който създава сцена в .json.gz формат, годна за прочитане от traytor

Рейтрайсърът поддържа текстури (включително HDR float), и те също се пакетират чрез base64 кодиране във сценовия файл.

В тази документация ще разгледаме много накратко работата на самия рейтрайсър, и ще обърнем повече внимание на паралелизацията. За инструкции за ползване се обърнете към страницата на проекта (<https://github.com/DexterLB/traytor>) и traytor --help, а за описание на имплементацията - към техническата документация (<https://godoc.org/github.com/DexterLB/traytor>).

3 Имплементация

Основният обект, който върши работа е [Raytracer](#). Важната функция е [Sample\(\)](#) - идеята е за всеки пиксел от картинаката да извикаме функцията [Raytrace\(\)](#). Тя ще намери цвета, който ще има този лъч:

- Изстреляме лъч от камерата, който отговаря на въпросните координати по экрана
- Гледаме къде (и дали) този лъч се пресича с триъгълната мрежа

([Mesh.Intersect\(\)](#)), като търсим триъгълника, който пресичаме в k-d дървото.

- Ако сме пресекли повърхност, знаем нейния материал. Извикваме [Material.Shade\(\)](#) за въпросната пресечна точка, и намираме получения цвят. Ако материалът е "лампа", това ще е дъното на рекурсията. Иначе материалът ще създаде нови лъчи и ще сметне цвета въз основа на техните цветове.

Така получаваме картичка, която е много шумна. Сега трябва да повторим същата операция много пъти(100 е добро голямо число, а 1000: още по-добро ¹).

3.1 Паралелизация по нишки ([traytor render](#))

- стартират се n на брой нишки, като всяка от тях инициализира свой [Raytracer](#) обект, със случайно генериран random seed, и своя празна картичка
- Всяка от нишките намаля с 1 споделен брояч ([SampleCounter](#)) и добавя sample към картинката. Повтаря тази операция, докато броячът не е станал 0.
- В момента, която някоя от нишките стане готова (общият брояч е станал 0, и тя е свършила да рендира своя sample), изпраща картинката си по общ канал.
- Главната нишка събира всички изображения, получени по канала в едно, и накрая го записва като .png

Така получаваме пълна паралелизация, и никоя от нишките не чака нищо от друга преди последния момент, в който се събира резултатът.

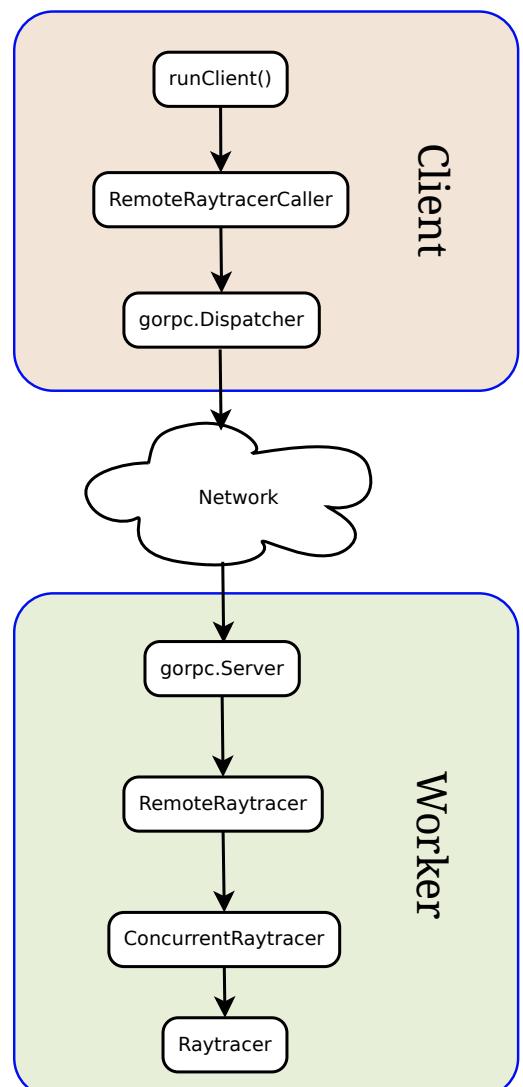
¹потвърдено от [ас. Росен Николов](#)

3.2 Паралелизация по машини ([traytor worker](#), [traytor client](#))

Тук имаме един клиент, комуникиращ чрез RPC библиотеката [gorpc](#) с много работници. Работниците слушат на някой TCP порт, а клиентът се свързва към тях, дава им работа и прибира резултата.

Имаме клас [RemoteRaytracerCaller](#) от страна на клиента, който вика функциите от [RemoteRaytracer](#) от страна на сървъра.

- Клиентът стартира лека нишка за всеки от работниците и извършва комуникацията с него в нея.
- Клиентът изпраща сцената на всеки от работниците ([LoadScene\(\)](#)). При получаване, работникът инициализира сцената (строи k-d дървото и т.н.)
 - В момента, в който сме свършили да пращаме сцена на някой работник, стартираме толкова нишки за този клиент, колкото заявки може да получава (потребителски параметър `max-requests`) - добре е да са повече, отколкото реалните нишки на работникът, за да може да има "изчакващи заявки": така когато работникът свърши срендитрането на един sample, започва следващия без да чака резултатът от предния да се върне до клиента.
 - Започваме цикъл, в който правим извиквания на [Sample\(\)](#). Както при паралелизацията по нишки, имаме споделен брояч, който намаляваме, и по който разбираме кога да спрем да раздаваме работа на работниците.



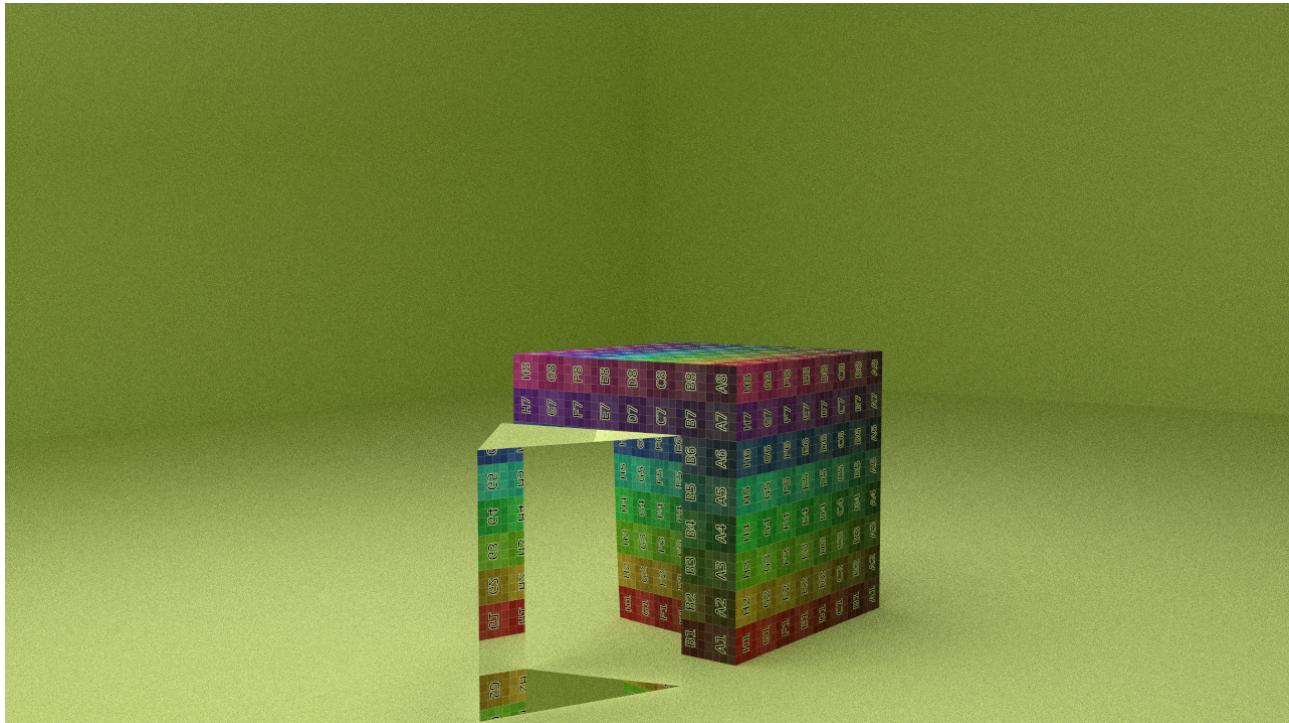
- От страната на работника, имаме `ConcurrentRaytracer`, който съдържа буфер с толкова `Raytracer` обекти и съответните им картички, колкото са нишките, които е поискал потребителя. Той се грижи винаги да се изпълняват не-повече от толкова `Sample()` функции едновременно, и всички едновременни извиквания да са върху различни `Raytracer` обекти, докато останалите заявки чакат в опашка. Това се реализира чрез буферираните канали в Go.
- Когато броячът стигне 0, и някой от работниците свърши със всички си текущи заявки, клиентът извика `GetImage()` на този работник. От страната на работника отделните картички се съединяват (те са толкова на брой, колкото са едновременните нишки), и като резултат се връща една картичка.
- При получаване на картичка от някой работник, клиентът я съединява с общата. Когато всички работници изпратят своя резултат, картичката се записва в .png

4 Тестови резултати

Тествателните резултати са правени върху сравнително лека сцена (1116 точки, 2046 триъгълника, 200 samples, 1280x720). Сцената на заглавната страница има 12245 триъгълника и отнема 41 минути и 21 секунди върху 15-те компютъра, използвани за тестове, с 500 samples / 1920x1080. Моделът на череп е предоставен в Public Domain от Cole Harris.

Тествателната сцена след пакетирането има размер 250kb, а сцената с черепа - 2.7mb.

Тествата сцена:



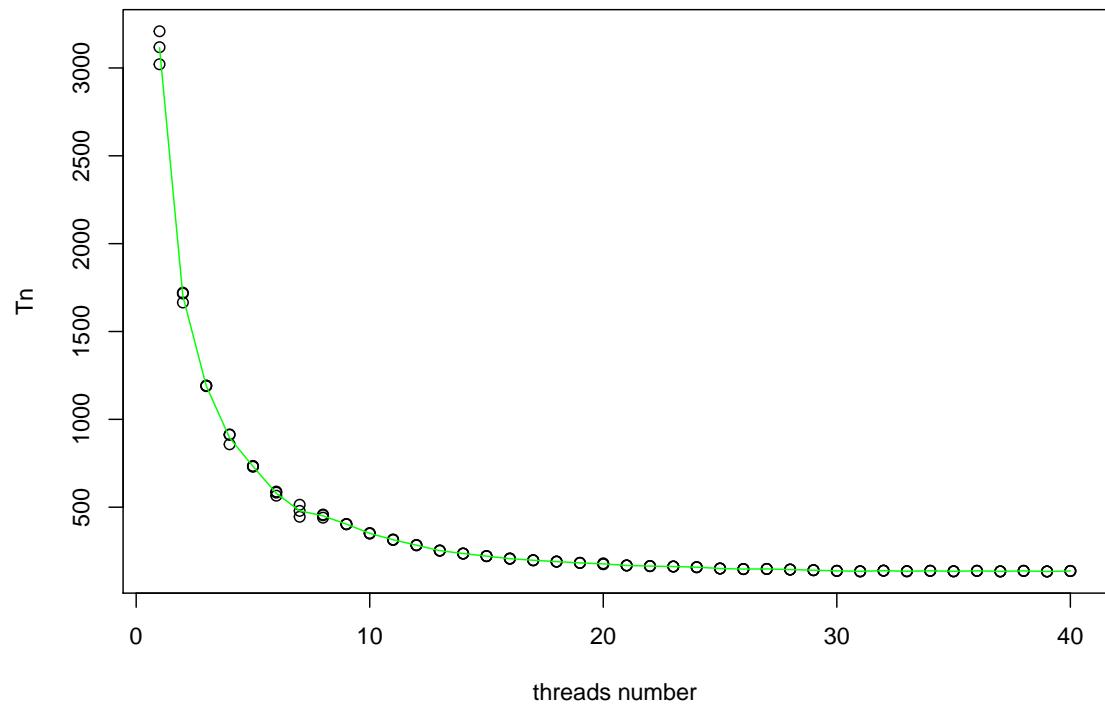
4.1 Паралелизация по нишки

Тук трябва да споменем, че нишките в Go са много добри и лесни за използване - по подразбиране са "леки нишки които се разпределят върху толкова на брой системни, колкото ядра имаме.

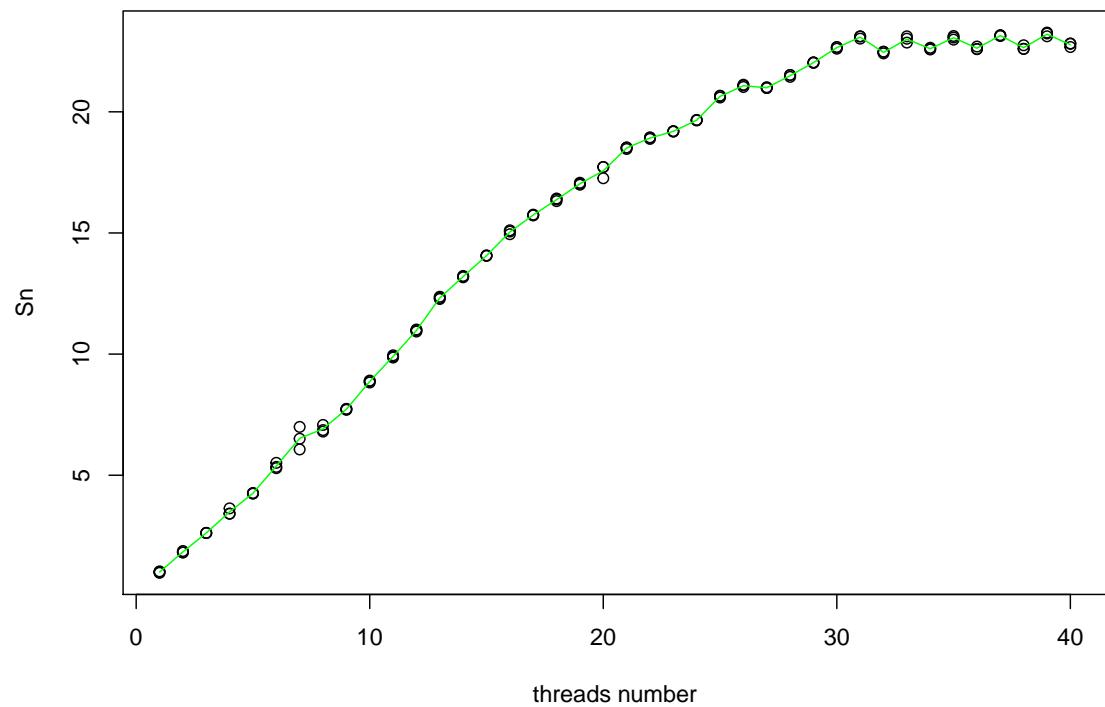


Тествете са извършени върху t5600.rmi.yaht.net (32 x Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz), операционна система CentOS Linux 7.2.1511 чрез ZSH скрипт, който за всеки брой нишки прави по 3 теста, и записва времената в .csv файл.

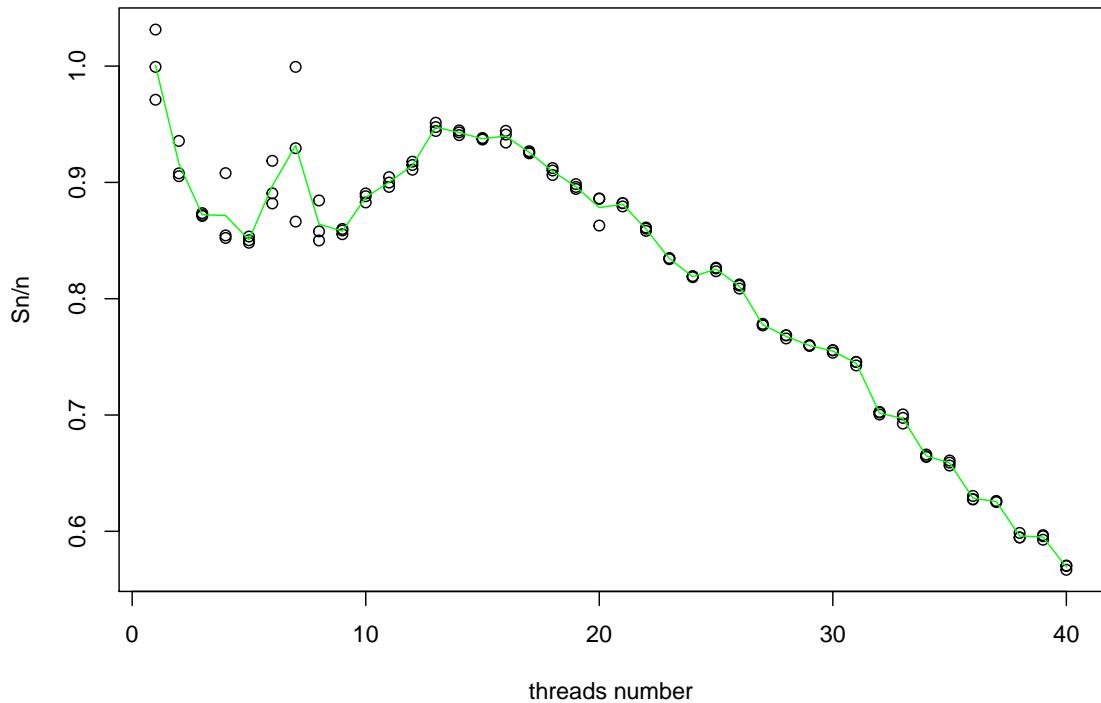
Време за изпълнение (T_n):



Ускорение ($S_n = T_1/T_n$):



Ефективност ($E_n = T_n/n$):



Наблюдения: При тестовете с малко нишки има шанс да има външно влияние върху резултатите (сървърът беше споделен между няколко човека, и има шанс да е имало колизии с някои тестове). Тъй като сървърът има 32 ядра, очевидно ефективността рязко пада след 32 на брой нишки.

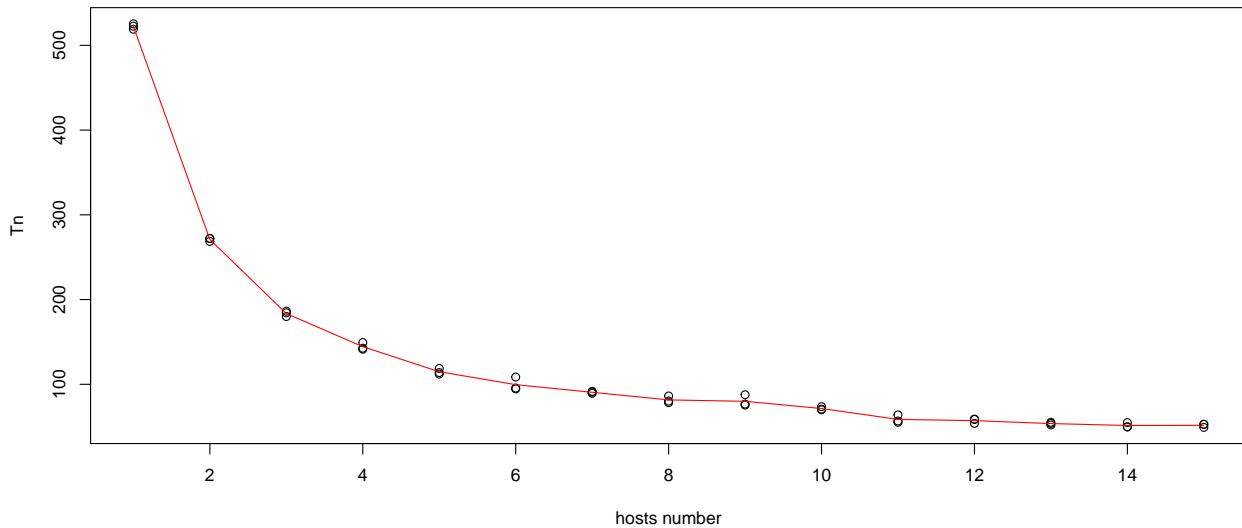
4.2 Паралелизация по машини

Тестовете са извършени върху 15 машини в зала 314 на ФМИ, всяка с по 4 ядра Intel(R) 2.8GHz, операционна система Windows 8.1 (workers) и клиент на Arch Linux, kernel 4.6.2

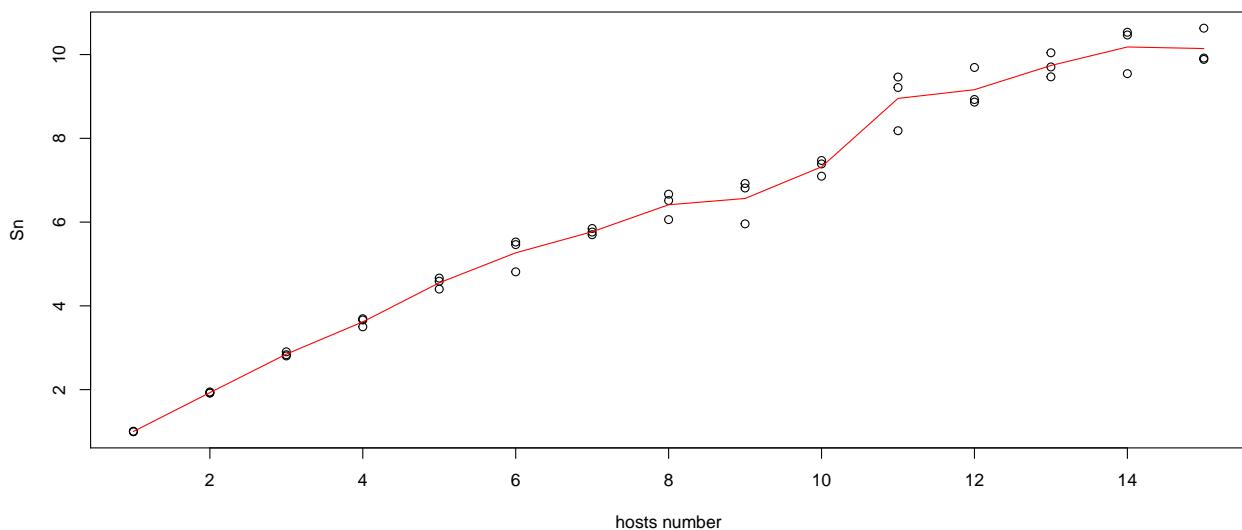
Главната пречка за ефективността беше мрежата в залата - почти едновременно пращане на 10mb картички от всички работници до клиента се оказа доста натоварващо за суича.

Тестващият скрипт прави по 3 теста за всеки брой машини (1-15), като на всеки тест избира n на брой произволни машини от всички.

Време за изпълнение (T_n):

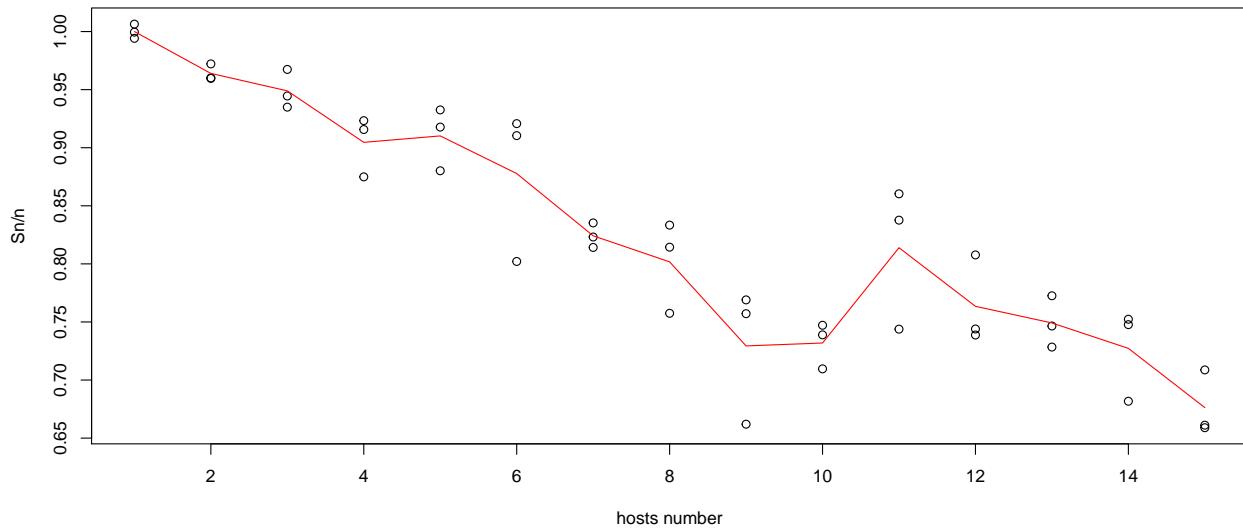


Ускорение ($S_n = T_1/T_n$):



Забележка: След теста с 10 машини се наложи рестартиране на суич, тъй като единият порт отказа да работи. Вероятно суичът се е претоварил - макар че трафикът е 0 през по-голямата част от времето, в някои моменти има голяма burst мрежова активност (работниците изпращат резултатите си на клиента)

Ефективност ($E_n = T_n/n$):



4.3 Таблични данни по нишки

Времената тук са осреднени (по 3 теста за всеки брой нишки)

| брой нишки | време за изпълнение | брой нишки | време за изпълнение |
|------------|---------------------|------------|---------------------|
| 1 | 3116.0656 | 21 | 168.3828 |
| 2 | 1700.8008 | 22 | 164.7116 |
| 3 | 1190.8138 | 23 | 162.3458 |
| 4 | 894.609 | 24 | 158.5271 |
| 5 | 732.6707 | 25 | 151.0038 |
| 6 | 579.1286 | 26 | 147.8263 |
| 7 | 479.4219 | 27 | 148.4224 |
| 8 | 450.8666 | 28 | 144.9769 |
| 9 | 403.514 | 29 | 141.4321 |
| 10 | 351.2554 | 30 | 137.5816 |
| 11 | 314.6988 | 31 | 134.9972 |
| 12 | 283.931 | 32 | 138.7647 |
| 13 | 252.9247 | 33 | 135.4976 |
| 14 | 236.0871 | 34 | 137.8525 |
| 15 | 221.5748 | 35 | 135.1356 |
| 16 | 207.21 | 36 | 137.7237 |
| 17 | 197.9808 | 37 | 134.621 |
| 18 | 190.3134 | 38 | 137.5868 |
| 19 | 182.9724 | 39 | 134.2713 |
| 20 | 177.4132 | 40 | 136.8628 |

4.4 Таблични данни по машини

| брой машини | време за изпълнение | машини (последен октет от IP адреса им) |
|-------------|---------------------|---|
| 1 | 522.592001792 | 129 |
| 1 | 525.40900608 | 139 |
| 1 | 519.013455104 | 136 |
| 2 | 268.649498112 | 144,140 |
| 2 | 272.02413824 | 145,142 |
| 2 | 272.150567936 | 145,129 |
| 3 | 186.231294976 | 137,141,146 |
| 3 | 184.337114624 | 136,142,140 |
| 3 | 179.980595456 | 143,134,140 |
| 4 | 142.613297152 | 146,130,135,139 |
| 4 | 141.431089664 | 136,145,135,139 |
| 4 | 149.255761152 | 143,140,137,129 |
| 5 | 118.692458496 | 144,130,137,129,146 |
| 5 | 112.023823104 | 143,139,132,134,140 |
| 5 | 113.827926528 | 140,132,143,129,136 |
| 6 | 94.550666496 | 132,146,143,141,130,135 |
| 6 | 108.538273792 | 134,129,143,135,141,142 |
| 6 | 95.622235904 | 145,130,137,139,132,140 |
| 7 | 387.322793472 | 146,142,136,143,135,144,140 |
| 7 | 89.336597248 | 132,137,142,134,146,135,141 |
| 7 | 91.65369472 | 132,136,145,139,143,141,130 |
| 8 | 80.175658752 | 143,132,135,140,141,134,146,142 |
| 8 | 78.349268736 | 142,132,130,134,143,135,145,140 |
| 8 | 86.200088064 | 135,134,142,137,141,136,143,129 |
| 9 | 76.654398464 | 143,137,129,140,141,136,139,134,146 |
| 9 | 87.671195648 | 140,134,145,135,139,141,136,146,144 |
| 9 | 75.471955456 | 135,134,146,141,142,139,129,136,137 |
| 10 | 73.608510208 | 140,130,141,144,146,143,129,134,142,145 |
| 10 | 70.692721664 | 146,140,143,142,135,134,130,129,144,139 |
| 10 | 69.910379776 | 135,146,137,136,134,139,129,132,130,142 |
| 11 | 55.195927552 | 144,130,132,146,136,142,137,139,145,134,140 |
| 11 | 63.840565504 | 141,142,136,129,145,143,139,135,140,134,130 |
| 11 | 56.68986624 | 144,143,146,145,141,134,129,130,139,140,135 |
| 12 | 53.890988032 | 144,141,130,129,139,137,143,146,135,140,142,136 |
| 12 | 58.921648128 | 129,143,144,141,134,130,146,137,132,136,139,140 |
| 12 | 58.510476032 | 146,137,136,139,145,140,132,135,130,134,142,129 |
| 13 | 52.009936384 | 134,139,145,141,146,135,144,143,137,130,140,132,129 |
| 13 | 53.823009536 | 136,129,144,141,135,142,132,145,134,143,146,139,130 |
| 13 | 55.163354624 | 137,145,129,139,130,136,142,143,144,146,140,134,141 |
| 14 | 49.592190464 | 136,134,139,130,142,141,146,143,140,145,129,135,132,144 |
| 14 | 49.900462592 | 142,144,146,143,140,135,132,134,137,136,130,145,129,141 |
| 14 | 54.727507968 | 139,140,132,134,141,137,146,129,142,144,143,135,145,136 |
| 15 | 52.844302592 | 130,134,132,129,144,137,140,145,141,139,136,143,135,146,142 |
| 15 | 49.135457024 | 144,146,136,134,129,139,140,143,142,132,137,135,141,145,130 |
| 15 | 52.671005184 | 139,142,146,130,129,132,144,136,145,135,143,141,137,134,140 |