Софийски университет “Св. Климент Охридски”

Факултет по математика и информатика

Проект по Системи за паралелна обработка

Обхождане на граф в дълбочина

5 юли 2017 г.

Изготвил:
Никола Стоянов
КН, курс 3, група 5
ф.н. 81151

Ръководител:
ас. Христо Христов

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 Алгоритъм

Решението на задачата може да се раздели на две независими части - генерирането на графа и обхождането му.

2.1 Генериране на графа

Понеже се иска да генерираме случаен граф, т.е. без някаква определена структура, задачата може да се реши с просто прилагане на принципа паралелизъм по данни. Ако трябва да работим с n нишки, разделяме върховете на графа на n равни части и във всяка част генерираме $\frac{1}{n}$-та от ребрата. Всяка нишка може да работи в отделна част и не е необходима допълнителна синхронизация между тях.

2.2 Обхождане на графа

За случая с една нишка се използва стандартния последователен dfs алгоритъм. Това служи за база за сравнение. За многонишковия вариант с K нишки първоначално си създаваме K на брой задачи и ги добавяме в thread pool-а. Всяка задача изпълнява леко модифицирана версия на стандартния dfs алгоритъм, започвайки от първия все-още необходен връх на графа. Всяка задача намира едно дърво на обхождане на свързана компонента от графа. Когато задача приключи се създава нова, докато не бъдат обходени всички върхове от графа.

За синхронизация между нишките се използва `used` масив от atomic променливи. Когато задача поиска даден връх се използва `CompareAndSwap` инструкция за да се отбележи, че върха е използван. При успех задачата става собственик на върха и може да го добави към стека си и полученото дърво на обхождането. При неуспех значи, че някоя друга задача я е изпреварила и е получила собственост над върха. Тогава върха се пропуска и се продължава с остатъка от алгоритъма. `CompareAndSwap` инструкцията гарантира, че само една задача може да получи собственост над даден връх.

При така показания алгоритъм всяка задача работи само с върхове, които е отбелязала за собствени. Това означава, че решението ще се различава от това, което бихме получили ако пуснем последователния dfs алгоритъм върху графа. По-точно върхове, които са част от една свързана компонента в графа, могат да бъдат разпределени в няколко различни дървета на обхождане. Това зависи от K - броят нишки върху които разпределяме задачата, както и поредността в която се изпълняват `CompareAndSwap` инструкциите и при по-голям брой нишки се увеличава разпокъсването.

Възможно е алгоритъма да се модифицира, така че резултатът да е същия, като от последователен dfs. Това се постига като си дефинираме подреждане на върховете на графа и в `used` масива запазваме кой връх е собственик. Също така се поддържа прехвърляне на собствеността от връх с по-висок номер към връх с по-нисък, като при това съответното поддърво на обхождането в оригиналния собственик става невалидно. Тази инвализация може да стане с линейно обхождане на резултата след приключване на алгоритъма, т.е. не е необходима синхронизация за нея. Така получения алгоритъм ще има идентичен резултат на последователния dfs, но има проблема, че по-голямата част от информацията, която се пресмята, по-късно става невалидна. При това значително се забавя алгоритъма дотолкова, че е по-бавен от еднонишковия вариант. Увеличаването на броя нишки също не помага, защото така рязко се увеличава и количеството информация, която се инвалидира.

3 Реализация

За реализация на проекта е използван езикът Rust. Използвани са следните външни библиотеки:

- `bit-vec` - bitmap структура от данни
- `clap` - обработка на command line аргументи
- `crossbeam` - lock-free структури от данни
- `rand` - генератори на случайни числа

Програмата поддържа следните command line аргументи:

```
USAGE:
    paralel-dfs [OPTIONS]

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

OPTIONS:
    -t, --threads <N>     Number of threads to use
    -i, --input <FILE>    File to read graph data from
    -n, --vertices <N>    Generate a graph with N vertices
    -m, --edges <N>       Generate a graph with N edges
```

Thread pool

За разпределяне на работата по множество нишки се използва ръчна имплементация на thread pool. Имплементацията поддържа само най-базовите функции като създаване на n нишки и добавяне на задачи за изпълнение в опашка. За опашката се използва готова имплементация на work-stealing queue от `crossbeam`.

Thread pool-а не поддържа връщане на резултат от изпълнението на задачата. За тази цел се използват канали (chanels, multi-producer single-consumer lock-free message queues) от стандартната библиотека на Rust. Те позволяват еднопосочна комуникация между нишки.

Генериране на графа

Поддържат се различни структури от данни за представяне на графа. Поддържат се списъци на наследство (Adjacency Lists), които позволяват генериране на ориентиран граф. Не може да се генерира неориентиран, защото за това ще се наложи значителна синхронизация между нишките. Имаше идея да се добави матрица на съседство (Adjacency Matrix), която да поддържа генериране и на ориентиран и на неориентиран граф, но това не е имплементирано. Всички представяния имплементират общ интерфейс `Graph`.

При генерирането на графа се използват генератори за случайни числа от библиотеката `rand`. За да се подобри скоростта за всяка нишка се създава отделен генератор.

4 Резултати

Всички резултати са измерени на 2x Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20GHz (Octacore CPU) с Hyperthreading. За входни данни са използвани:

- брой върхове N = 24 мил.
- брой ребра M = 196 мил.

Резултатите от отделните алгоритми могат да се видят на фигура 1.

Генериране на ориентиран граф

Генерирането е прост алгоритъм, който може лесно да бъде паралелизиран на ниво данни, поради което показва високи нива на ускорение. Основното забавяне при по-висок брой нишни се дължи на това, че някои нишки свършват работата си по-бързо от други, но това няма как да се избегне, защото трябва да изчакаме целият граф да бъде генериран преди да продължим нататъка.

Threads	Time (s)	S_p	E_p
1	72.08	1.00	1.00
2	37.16	1.94	0.97
3	22.09	3.26	1.09
4	17.51	4.12	1.03
6	11.17	6.45	1.08
8	9.01	8.00	1.00
10	7.92	9.10	0.91
12	6.36	11.33	0.94
14	6.06	11.89	0.85
16	5.19	13.89	0.87
20	5.37	13.42	0.67
24	4.71	15.30	0.64
28	4.4	16.38	0.59
32	3.93	18.34	0.57

Обхождане на графа

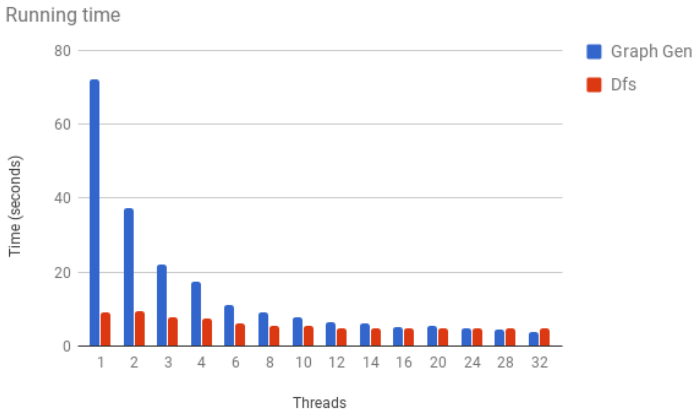
Обхождането е значително по-сложно затова постига много ниско ускорение.

Threads	Time (s)	S_p	E_p
1	9.23	1.00	1.00
2	9.54	0.97	0.48
3	7.7	1.20	0.40
4	7.3	1.26	0.32
6	6.1	1.51	0.25
8	5.6	1.65	0.21
10	5.4	1.71	0.17
12	4.95	1.86	0.16
14	4.82	1.91	0.14
16	4.68	1.97	0.12
20	4.9	1.88	0.09
24	4.89	1.89	0.08
28	4.91	1.88	0.07
32	4.65	1.98	0.06

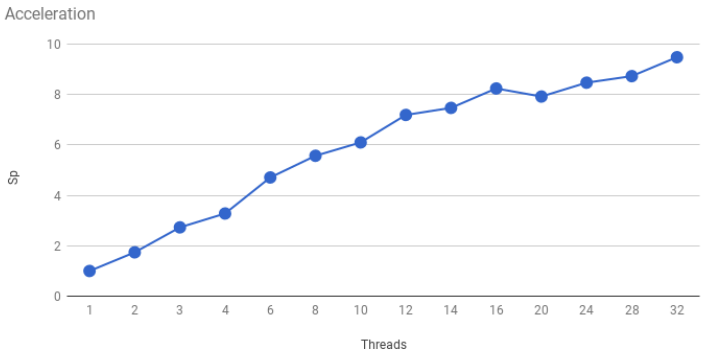
Финална оценка

Графики на ускорението и ефикасността на цялата програма могат да се видят на фигури 2 и 3.

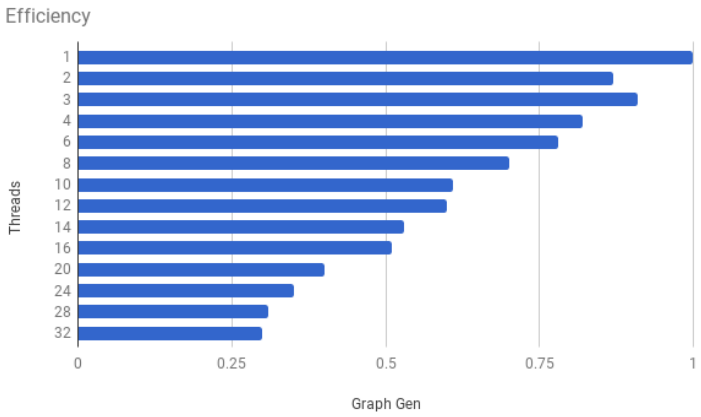
Threads	Time (s)	S_p	E_p
1	81.31	1.00	1.00
2	46.7	1.74	0.87
3	29.79	2.73	0.91
4	24.81	3.28	0.82
6	17.27	4.71	0.78
8	14.61	5.57	0.70
10	13.32	6.10	0.61
12	11.31	7.19	0.60
14	10.88	7.47	0.53
16	9.87	8.24	0.51
20	10.27	7.92	0.40
24	9.6	8.47	0.35
28	9.31	8.73	0.31
32	8.58	9.48	0.30



Фигура 1: Време за изпълнение на отделните части от алгоритъма



Фигура 2: Графика на ускорението



Фигура 3: Графика на ефикасността