Софийски университет "Св. Климент Охридски"

Факултет по математика и информатика

Проект по Системи за паралелна обработка

Обхождане на граф в дълбочина

2 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 - създаването на графа и обхождането му. Графа може да бъде създаден или като бъде прочетен от файл, който го описва, или като бъде генериран на случаен принцип.

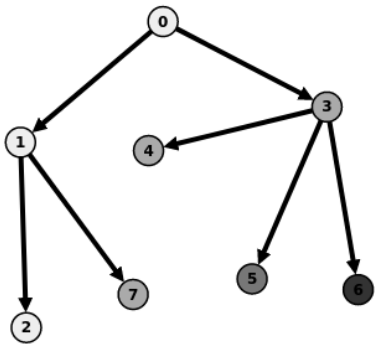
Прочитане на графа. Това не е реализирано, макар че се изискваше от задачата. :(

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.

Обхождане на графа. За случая с една нишка се използва стандартния последователен dfs алгоритъм. Това служи за база за сравнение. Основната трудност при многонишковия случай е, че трябва да запазим същата последователност на обхождане на върховете. За целта използваме една нишка, която прави обхождане в дълбочина, но без backtracking. Тоест започва от начален връх, преминава към първия му съсед, след това към първия съсед на съседа и т.н. докато не стигне до връх без повече необходени съседи. През всеки посетен връх, всички съседи освен първия се подават на допълнителни нишки, които правят същото обхождане. Това е визуализирано на фигура 1.

Идеята е докато една нишка обхожда графа, останалите да пробват останалите пътища. Ако в бъдеще първата нишка стигне до върхове обходени от някоя друга, то тези върхове трябва да бъдат обходени отново, защото е намерен "по-къс" път. Но ако това не се случи значи е намерен път, който е част от решението на задачата и в последователния алгоритъм би бил намерен при backtracking-а.

Този начин на обхождане налага за всеки връх да се пази не просто дали е обходен, ами от коя нишка е обходен. Освен това всяка проверка дали връх е вече посещаван трябва да се синхронизира с останалите нишки.



Фигура 1: Разпределяне на търсенето по нишки. Върховете с един и същ цвят се обхождат от една и съща нишка.

3 Реализация

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

- `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 [FLAGS] [OPTIONS]

FLAGS:
        --undirected    Generate an undirected graph (defaults to directed)
    -h, --help          Prints help information
```

```
    -V, --version       Prints version information

OPTIONS:
    -t, --threads <N>      Number of threads to use
    -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

Каква структура от данни се използва за представяне на графа зависи от това дали е ориентиран или неориентиран. Ако е ориентиран се използва списък на съседство, в противен случай матрица на съседство. С изключение на това имплементацията е почти идентична и следва алгоритъма представен в предишната секция. Освен това и двете представяния имплементират общ интерфейс `Graph`, което означава, че това не се отразява на останалата част от кода.

Забележка: Не е необходимо да се използва списък на съседство за неориентирания случай, и матрица би работила също толкова добре, но реших че е полезно да има повече варианти за тестване. Обратното обаче не е съвсем вярно, защото за генерирането на ориентиран граф със списъци на наследство най-вероятно ще се наложи значителна синхронизация между нишките.

Обхождане на графа

Дефинираме си дължина на пътя като колко съседа трябва да прескочим в ширина. Използвайки фигура 1 за илюстрация пътят от 3 до 4 има дължина 0, пътят от 3 до 5 има дължина 1, а пътят от 3 до 6 има дължина 2. Пътищата от 0 до 2, 7, 4, 5, 6 имат дължини съответно 0, 1, 1, 2, 3.

За всяка дължина си създаваме нова задача и я добавяме в thread pool-а. Понеже не знаем каква ще е максималната дължина която ще достигнем предварително, допълнителни задачи се добавят към thread pool-а по време на изпълнението. Всяка задача обхожда върхове само с определена дължина на пътя. За комуникация между задачите се използват channel-и, като всяка задача си създава собствен канал, запазва си приемащата част за себе си и размножава и разпраща изпращащите части до другите задачи. Всяка задача си има собствен стек (от един елемент) за върховете които обхожда. Когато обходи връх добавя първия му съсед към собствения си стек, а останалите разпраща на други задачи като им изпраща съобщение по channel-а.

Вижда се, че задача може да приема съобщения само от задачи с по-малка дължина. Това означава, че задачата може да приключи само ако всички преди нея са приключили. Поради това е важно задачите да бъдат добавени и изпълнение в правилния ред в thread pool-а, иначе ще има deadlock.

За да се следи кои върхове са били посетени от кои задачи се използва следната структура.

```
struct Visit {
    root_vertex: usize,
    path_len: usize,
    parent_vertex: usize,
}
```

Масив от такива обекти (по един за всеки връх) е достъпен като споделена памет от всички задачи. Два обекта могат да се сравняват като `a < b <=> (a.root_vertex, a.path_len, a.parent_vertex) < (b.root_vertex, b.path_len, b.parent_vertex)` (т.е. лексикографска наредба, *(ггрр СЕП)*). Така като някоя задача стигне до посещаван връх проверява дали пътят, по който вмомента обхожда е "по-кратък" от предишно намерените.

Оптимизации

Една от основните оптимизации, които се наложи да направя, беше върху структурата `Visit`. В първоначалната си форма споделеният масив имаше тип `Arc<Vec<RwLock<Visit>>>`, където `Arc` е Atomic Reference Counter, `Vec` е Vector, а `RwLock` е Read-Write Lock, тоест синхронизационен обект който позволява едновременно множестово достъпи за четене или един достъп за писане до променливата. Това беше необходимо, защото трябва съдържанието на масива да може да се променя от много нишки едновременно, но означаваше че всеки път когато трябва да се провери кой път е по-кратък се налага да се заключи променливата. А такива заключвания стават постоянно, защото това се намира на hot path-а на програмата.

Решението беше да се пренаправи структурата `Visit` до

```
struct Visit {
    root_vertex: AtomicUsize,
    path_len: AtomicUsize,
    parent_vertex: AtomicUsize,
    lock: Mutex<()>,
}
```

Промяната е, че вкарваме синхронизацията вътре във класа, което значи, че масивът може да стане `Arc<Vec<Visit>>`. В този случай можем да правим проверка дали обект е по-малък от tuple `(root, path_len, parent)` (които са нормални nonatomic integer-и) без да заключваме обекта.

Това звучи опасно на пръв поглед, защото прочитането на три отделни AtomicUsize променливи не е atomic операция, и можем да видим стари данни, нови данни или мешаница от двете. Но въпреки това знаем, че стойностите, които `Visit` обект приема са монотонно намаляващи, защото такъв обект може да се презапише единствено от стойност по-малка от текущата му. Поради това независимо какви стойности прочетем при load-а, винаги можем да отговорим правилно на въпрос `a < b` с:

- да, сигурни сме, че `a < b`
- не знам, може `a < b`, а може и `a >= b`. Заключи променливата и провери отново.

Дори и да не получаваме категоричен отговор подобна проверка е много полезна, защото ако върне "да" значи няма нужда да продължаваме по-нататък с текущия връх. И въпреки, че false positive отговорите увеличават броя пътища които трябва да проверим, премахването на голяма част от lock-овете води до над двойно намаляне във времето за изпълнение.

4 Резултати

Генериране на ориентиран граф

Threads	Time (s)	S_p	E_p
1	45.29	1.0000	1.0000
2	22.5	2.0129	1.0064
3	15.51	2.9201	0.9734
4	11.04	4.1024	1.0256
6	7.55	5.9987	0.9998
8	6.11	7.4124	0.9266
12	4.06	11.1552	0.9296
14	3.71	12.2075	0.8720
16	3.55	12.7577	0.7974
20	3.28	13.8079	0.6904
24	2.84	15.9472	0.6645

Обхождане на графа

Threads	Time (s)	S_p	E_p
1	7.16	1.0000	1.000
2	98.1	0.0730	0.0365
3	79.93	0.0896	0.0299
4	59.55	0.1202	0.0301
6	42.65	0.1679	0.0280
8	30.67	0.2335	0.0292
12	28.42	0.2519	0.0210
14	26.86	0.2666	0.0190
16	27.3	0.2623	0.0164
20	25.39	0.2820	0.0141
24	25.41	0.2818	0.0117