Обхождане на граф в дълбочина

Проект по Системи за паралелна обработка

30 юни 2018 г.

Изготвил: Никола Стоянов
КН, IV курс, ф.н. 81151

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

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

- `spin` - имплементация на Spin Lock
- `structopt` - обработка на аргументи от командния ред
- `rand` - генератори на случайни числа
- `rayon` - библиотека за паралелна обработка на данни

Проектът е реализиран като библиотека, която предоставя функционалността от изискванията. Всички измервания са правени директно върху съответните функции, използвайки benchmarking инструментите на езика. Предоставено е и просто приложение, което приема аргументи от командния ред и извиква съответните функции.

Изискването за въвеждане на граф от входен файл не е имплементирано.

3 Генериране на графа

Решението на задачата може да се раздели на две независими части - генерирането на графа и обхождането му. Графа се запазва в списъци на съседство, защото реших, че това ще е най-удачно откъм необходима памет.

3.1 Ориентиран граф

Генерирането на ориентиран граф е лесна задача за разпарелявване ако се използва принципа паралелизъм по данни. Изпробвани два подхода:

- Ако разполагаме с t на брой нишки, то разделяме върховете на графа на $\frac{n}{t}$ равни части и във всяка част генерираме $\frac{m}{t}$ ребра. Всяка нишка работи върху отделна част от графа.
- Разделяме графа на части с фиксирана големина от K върхове и $\frac{m}{K}$ ребра. Частите се добавят като задачи в опашка и thread pool-а се грижи за разпределянето им по наличните нишки. В кода за стойност на K е използвано 128.

Двата подхода са разгледани главно за да се сравни ръчното разпределяне на задачата (първи подход) с автоматичното разпределяне предоставяно от rayon (втори подход).

3.2 Неориентиран граф

Генерирането на неориентиран граф е по-сложна задача, защото за всяко добавено ребро *(u, v)*, трябва да се добави и обратното ребро *(v, u)*. Това не би било проблем ако се използваше матрица на съседство, но решението графа да се запазва в списъци на съседство усложнява проблема.

Първата стъпка е да се генерират ребрата *(u, v)* така, че $u > v$. Използва се втория 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. За това е нужен конкурентен достъп до структурата. Ако в даден момент се опитваме едновременно да обходим ребрата *(a, v_i)* и да добавим реброто *(a, b)* ще има конфликт между четец и писател за вектора `lists[a]`. Аналогично ако се опитаме да добавим ребра *(a, b)* и *(a, c)* ще имаме конфликт между два писателя. Възможни решения са:

- Добавянето на огледални ребра се реализира последователно, използвайки само една нишка.
- Използва се механизъм за заключване, като отделните вектори се заключват поотделно. Изпробвано е с mutex на операционната система и spin lock. Добре е да се отбележи, че оригиналните ребра са от вида *(u, v)*, където $u > v$, поради което е невъзможно да се получи deadlock.
- Използва се допълнителна lock-free структура от данни, например опашка, към всеки вектор, в която се записват върховете които трябва

да се добавят. След обхождането върховете могат да се добавят без проблем. За съжаление този подход не е тестван поради бъг в имплементацията :(.

4 Обхождане на графа