Раскраска графа по ребрам

Попробуем покрасить графа. Покрасим ему ребра так, чтобы из одной вершины не торчало двух ребер одного цвета. Число цветов будет не меньше, чем максимальная степень вершины графа, то есть максимальное число ребер, инцидентных одной вершине. Сделаем это каким-нибудь простеньким жадным алгоритмом (см. в интернете про жадную раскраску и прочие раскраски, если интересно). Там вряд ли получится ровно минимум цветов, обычно на один-два больше. Но для наших целей это вполне приемлемо, несколько лишних цветов почти никак не влияют на производительность. Покрасили, получилось, как на картинке ниже.

Получилось, что если параллельно обрабатывать не все ребра, а ребра только одного цвета, то не будет пересечений по данным. У ребер одного цвета нет общих вершин, гонка исключена.

Значит, будем обрабатывать ребра по цветам по очереди. Между цветами обязательно барьер, иначе цвета наползут друг на друга, появятся общие вершины, возникнет гонка. Поменяем нумерацию ребер. Переупорядочим все ребра по цветам. Сначала будут идти все ребра одного цвета, потом другого, ... Все же понимают, как менять нумерацию? Если что, обсудим на занятии. Пусть получилось NC цветов (C – Color). Занесем диапазоны номеров каждого цвета в массив цветов C размера NC+1. C[i] – позиция начала i-го цвета в массиве ребер. С[NC] равен NE (число ребер).

Теперь получилось, что операция делается не за раз, а за некоторое количество тактов (у каждого цвета свой параллельный цикл) с синхронизацией после каждого такта. Это несколько снижает производительность, но это незаметно, если объем вычислений достаточно большой. Но это не главная проблема. Основная проблема в том, что мы снижаем повторную используемость данных в кэш. Раньше мы могли упорядочить ребра лексикографически, чтобы, обрабатывая ребра подряд, мы часто попадали в одну и ту же вершину (или близкие по номерам вершины) при записи результата. А тут, когда мы адресуемся в инцидентные ребру вершины, мы точно не попадем в одну и ту же вершину. Значит, число кэш-промахов будет заметно выше. Недостатки: хуже локальность доступа данных.

Декомпозиция, последовательная обработка интерфейсных граней Распределим работу между нитями статически, распределив вершины графа между параллельными потоками (нитями). Сопоставим потокам цвета, чтобы удобнее было рисовать. Получается, надо задать вершинам цвета. За такую декомпозицию отвечает какой-то внешний алгоритм, какая-то библиотека. Об этом более подробно в разделе 4.3.1. Этот алгоритм так распределит нам вершины графа, чтобы 1) разбиение было сбалансированным (число вершин одного цвета было примерно одинаковым); 2) чтобы количество ребер с вершинами разного цвета, aka edge-cut, было как можно меньше. Такие ребра, у которых вершинки разных цветов, назовем интерфейсными. Они нам и создают состояние гонки. Теперь по раскраске вершин покрасим ребра. Ребра, у которых обе вершины одного цвета, в этот цвет и выкрасим. Интерфейсные ребра будут отдельного цвета (на картинке – черный), которого не было в вершинах. Пример показан на картинке ниже.

Как и в предыдущем случае, переупорядочим все ребра по цветам. У ребер получилось NC цветов. Это NC равно NT+1, где NT – число нитей (T – Threads). Занесем диапазоны номеров ребер для каждого цвета в массив цветов C размера NC+1. В C[k] будет позиция начала k-го цвета в массиве всех ребер E, который мы упорядочили по цветам. Ребра k-го цвета будут в интервале ie = C[k], …, С[k+1]-1. Соответственно, в последнюю позицию С[NC] записываем число ребер NE. У каждой нити есть свой цвет. Но что делать с интерфейсом? А интерфейс пусть отдельно обработает какая-то одна нить в последовательном режиме.

Минусы: последовательная обработка интерфейса, деградация с ростом числа нитей. Если нитей мало, а ребер много, вес интерфейса будет небольшим, и мы получим приемлемое ускорение. Но когда нитей много, последовательная обработка интерфейса всё портит. Попробуем это исправить.