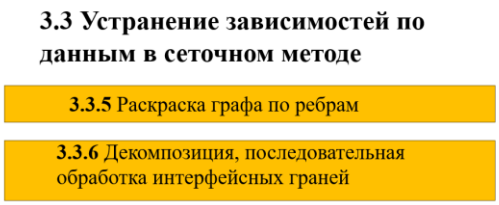


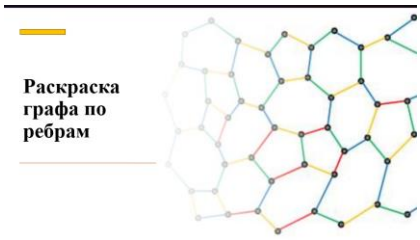
продолжаем Разговаривать об "Устранение зависимостей по данным в сеточном методе", который относится к параллельному вычислениям, где проблема разбивается на меньшие части, которые выполняются параллельно на разных вычислительных устройствах или ядрах. Они уже рассказали о первых четырех шагах, теперь я расскажу о пятом и шестом:

Раскраска графа по ребрам.

Декомпозиция, последовательная обработка интерфейсных граней.

Давайте рассмотрим каждый из них более подробно:

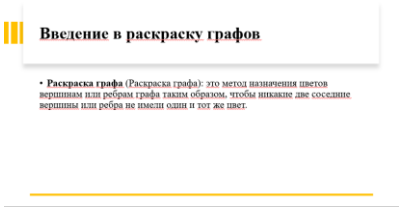
Move to the next slide.



Introduction:

Мы будем изучать идею "Раскраски графа по ребрам", что означает "Graph Coloring by Edges". Это понятие важно в параллельных вычислениях и помогает нам определить и выполнять задачи одновременно, не вызывая конфликтов.

Move to the next slide.

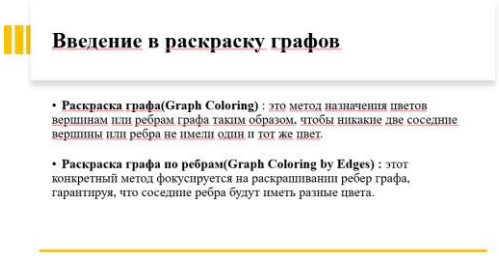


Definitions:

Мы знаем, что такое граф, но давайте быстро разберемся, что такое Раскраска графа:

Раскраска графа (Graph Coloring): Это как метод раскраски точек или ребра на картинке так, чтобы никакие две точки или ребра , которые соприкасаются, не имели одинакового цвета.

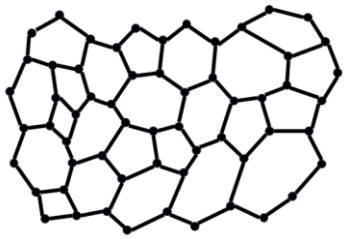
Move to the next slide.



Раскраска графа по ребрам (Graph Coloring by Edges): Этот специфический метод сконцентрирован на раскрашивании ребра на картинке так, чтобы соседние ребра имели разные цвета.

Move to the next slide.

Раскраска графа по рёбрам



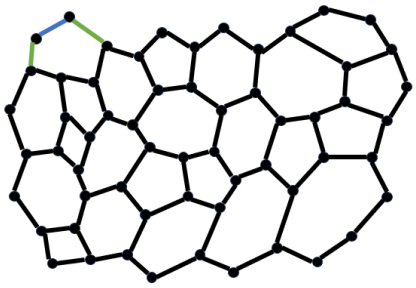
Представьте, что у нас есть этот граф, и мы хотим раскрасить его.

Давайте представим, что у нас есть граф и мы хотим Раскрасить его рёбрам, так, чтобы никакие две рёбра одного цвета не соединялись с одной и той же точкой.

Move to the next slide.

Вот как мы это делаем:

Раскраска графа по рёбрам



Начинаем с пустого списка покрашенных рёбро.

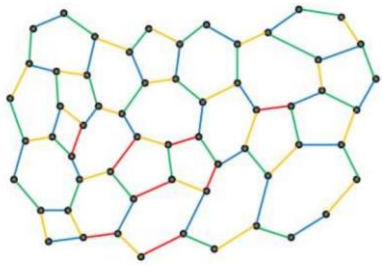
Выбираем рёбру, которую ещё не покрасили.

Даем этой рёбра самый маленький доступный цвет, который не использовался для соседних рёбро.

Повторяем шаги 2 и 3, пока все рёбра не будут покрашены разными цветами.

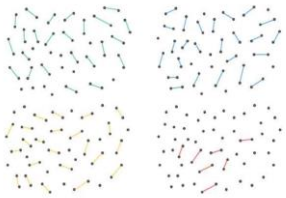
Количество разных цветов, которое нам понадобится, будет как минимум равно максимальному количеству рёбра, связанных с одной и той же точкой.

Move to the next slide.



И вот результат: мы покрасили граф, и вы можете увидеть его на картинке.

Move to the next slide.



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.

Затем, чтобы обрабатывать рёбра (части данных) более эффективно, мы делаем это по отдельным цветам, чтобы избежать ошибок. Мы сначала обрабатываем рёбра одного цвета, затем другого и так далее. Для того, чтобы не перепутать данные разных цветов, мы переупорядочиваем порядковые номера рёбер. Чтобы лучше понять, как это работает в программировании, мы создаём список диапазонов цветов, где C[i] указывает, с какого места начинаются данные цвета i в списке рёбер, а C[NC] - сколько всего рёбер в списке.

Теперь мы можем параллельно обрабатывать данные, используя инструмент OpenMP. Мы идём по списку цветов и, внутри каждого цвета, по его данным. После того как обработаем данные каждого цвета, мы делаем паузу, чтобы всё успело завершиться.

Move to the next slide.

Исходный код

```
#pragma omp parallel
{
    for(int ic=0; ic<NC/*Number of colors*/; ++ic){ // цикл по цветам
        #pragma omp for
        for(int ie=C[ic]; ie<C[ic+1]; ++ie){ // цикл по ребрам графа данного цвета
            double r = Calc(ie);
            X[E[ie].v[0]] += r;
            X[E[ie].v[1]] += r;
        }
        #pragma omp barrier // для наглядности (в omp for неявный барьер уже есть)
    }
}
```

Этот фрагмент кода представляет собой параллельную реализацию обработки цветов рёбер в контексте раскраски графа. Вот разбор кода и его назначения:

1. **#pragma omp parallel**: Эта строка указывает на начало параллельной области с использованием OpenMP (API параллельного программирования). Это означает, что следующий код будет выполняться параллельно несколькими потоками.

Move to the next slide.

2. **for(int ic=0; ic<NC/*Number of colors*/; ++ic)**:

   Этот цикл проходится по цветам, которые используются для раскрашивания графа. NC - это количество цветов. Каждый цвет обрабатывается отдельно одновременно.

   Move to the next slide.

3. **#pragma omp for**: Эта строка представляет собой директиву OpenMP, которая параллелизирует последующий цикл. Грани каждого цвета обрабатываются параллельно разными потоками.

   Move to the next slide.

4. **for(int ie=C[ic]; ie<C[ic+1]; ++ie)**:

   Внутри каждого цвета этот цикл проходит по всем рёбрам графа, которые относятся к текущему цвету. Массив C хранит информацию о местоположении каждого ребра для каждого цвета, а ie - это номер (порядковый номер) этого ребра.

   Move to the next slide.

5. **double r = Calc(ie);**:

   Для каждого ребра выполняется вычисление с использованием функции Calc, и результат сохраняется в variable **r**.

   Move to the next slide.

6. **X[E[ie].v[0]] += r;** and **X[E[ie].v[1]] += r;**:
   Мы берем результат, который мы посчитали, и прибавляем его к двум точкам (узлам), которые соединены между собой ребром. Этот шаг, скорее всего, включает в себя обновление информации, связанной с этими точками в графе.

   Move to the next slide.

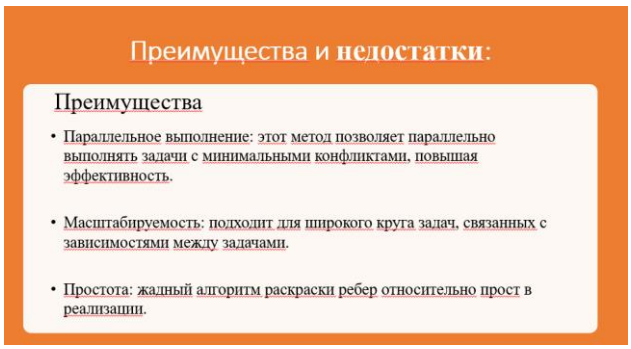
7. **#pragma omp barrier**:

   Эта команда помогает разным задачам в программе работать вместе. Она убеждается, что все задачи закончат свою работу с одним делом, прежде чем начнут следующее.

Этот метод делает так, чтобы компьютер более часто не мог найти нужную

информацию в своей памяти, что замедляет его работу. Раньше мы упорядочивали данные специальным образом, чтобы ускорить доступ к одним и тем же данным. Теперь мы делаем так, чтобы данные, к которым мы часто обращаемся, хранятся в разных частях памяти, и это приводит к большему числу медленных поисков в памяти.

Move to the next slide.



Итак, давайте теперь поговорим о преимуществах и недостатках.

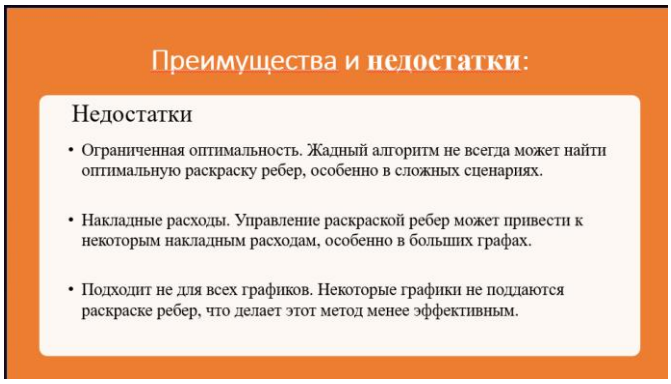
Преимущества:

1. Параллельное выполнение: Этот метод позволяет параллельное выполнение задач с минимизацией конфликтов, что повышает эффективность.

2. Масштабируемость: Он подходит для широкого спектра задач, включающих зависимости между задачами.

3. Простота: Алгоритм жадной раскраски рёбер относительно прост в реализации.

Move to the next slide.



Недостатки:

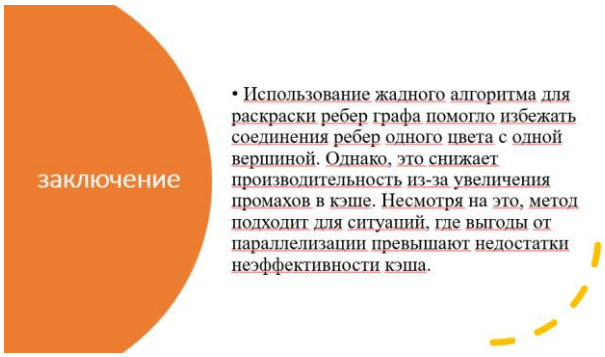
1. Ограниченная оптимальность: Жадный алгоритм не всегда способен найти оптимальное раскраску рёбер, особенно в сложных сценариях.

2. Накладные расходы: Управление раскраской рёбер может внести некоторые накладные расходы, особенно в случае больших графов.

3. Не подходит для всех графов: Некоторые графы могут быть не подходящими для раскраски рёбер, что делает эту технику менее эффективной.

Move to the next slide.

Conclusion



В заключение, наш метод раскраски рёбер графа с использованием жадного алгоритма доказал свою эффективность в обеспечении того, чтобы ни одно из двух рёбер одного цвета не соединяло одну и ту же вершину. Однако этот подход вводит определённые компромиссы. Путём обработки рёбер разных цветов отдельно и переупорядочивания нумерации рёбер, мы можем эффективно параллелизировать вычисления, предотвращая конфликты данных между цветами. Тем не менее, это сопровождается ухудшением локальности доступа к данным и, как следствие, увеличением числа промахов кеша.

Несмотря на то, что этот подход может привести к небольшому снижению производительности, он остаётся приемлемым решением для ситуаций, где преимущества параллелизации превышают недостатки неэффективности кеша. В зависимости от конкретной задачи и объёма вычислений, он может быть ценным методом оптимизации алгоритмов раскраски графов и других параллельных вычислительных задач.