Параллельная реализация алгоритмов Борувки и Дейкстры

Губарьков Никита
6371

Моей задачей было написать параллельную реализацию алгоритмов Борувки и Дейкстры

Суть алгоритмов:

Дейкстра:
Дан граф, выбрана одна вершина. Требуется найти дерево кратчайших путей из выбранной вершины во все остальные

Борувка:
Дан граф, требуется найти его минимальное остовное дерево.
Остовное дерево - подграф, содержащий все вершины исходного графа и не имеющий циклов.
Минимальное о.д. - о.д. с минимальной суммой весов ребер.

Какие задачи мне пришлось решить в ходе работы

Дейкстра:

- Реализация алгоритма, код для master-потока и для потоков-кластеров

Борувка:

- Реализация алгоритма, один код для потоков-ядер
- Реализация функционала find-union* для атомарного объединения компонент графа

Прикладные задачи:

- Разбиение графа на связные подграфы (поиск в глубину на небольших графах и в ширину на больших для предотвращения переполнения стека)
- Реализация удобных инетрфейсов для работы с графами: структуры поддерживающие итерацию по ребрам, вершинам, соседям вершины, итд, структуры эффективны по памяти и производительности
- Реализация однопоточного или параллельного копирования больших графов для последующей обработки алгоритмом

* под атомарным find-union подразумевается поиск компоненты связности по вершине и объединение компонент в многопоточной среде без использования блокировок

Краткий гайд в многопоточное программирование

* i++ состоит из чтения переменной, увеличения на 1 и записи:
  1. int t=i;
  2. t=t+1;
  3. i=t;

Допустим, у нас есть переменная (int i=0) и 2 потока, которые инкрементируют ее (i++)*
Рассмотрим, как это будет работать в 3х случаях (по возрастанию сложности кода):

"В лоб"

2 потока вызывают i++:

1. Поток А читает i (0)
2. Поток В читает i (0)
3. Поток А рассчитывает новое значение (1)
4. Поток А записывает 1 в i
5. Поток В рассчитывает новое значение (1)
6. Поток В записывает 1 в i

Итого, мы получаем i=1
А должно быть i=2

Блокировка

Потоки блокируют выполнение друг друга чтобы одновременно не изменять значение i:

1. А пытается захватить монитор (блокировку), удачно
2. В пытается захватить монитор, неудачно, ждет освобождения
3. А увеличивает i
4. А освобождает монитор
5. В захватывает монитор
6. В увеличивает i
7. В освобождает монитор

Итого, i=2

Lock-free

Потоки вычисляют новые значения и пытаются обновить i пока им это не удастся:

1. А читает i (0) и вычисляет i+1 (1)
2. В читает i (0) и вычисляет i+1 (1)
3. А пытается записать 1, если текущее значение i-0, удачно
4. В пытается записать 1, если текущее значение i-0, неудачно
5. В повторяет все заново: читает i (1), вычисляет i+1 (2), пытается записать если текущее значение i-1, удачно

Итого, i=2

В моей работе, все алгоритмы реализованы в lock-free, т.к. этот способ исключает блокировки и дает огромный прирост к производительности

Дейкстра (Dijkstra) Классическая версия

Псевдокод:

/*Каждой вершине вершине соответствует метка (она означает текущее рассчитанное расстояние
до этой вершины, изначально метка исходной вершины равна нулю, всех остальных-бесконечности.
Также каждая вершина может быть помечена как "обработанная", исходная помечена изначально*/

Пока не все вершины обработаны:

Выбираем вершину с минимальной меткой

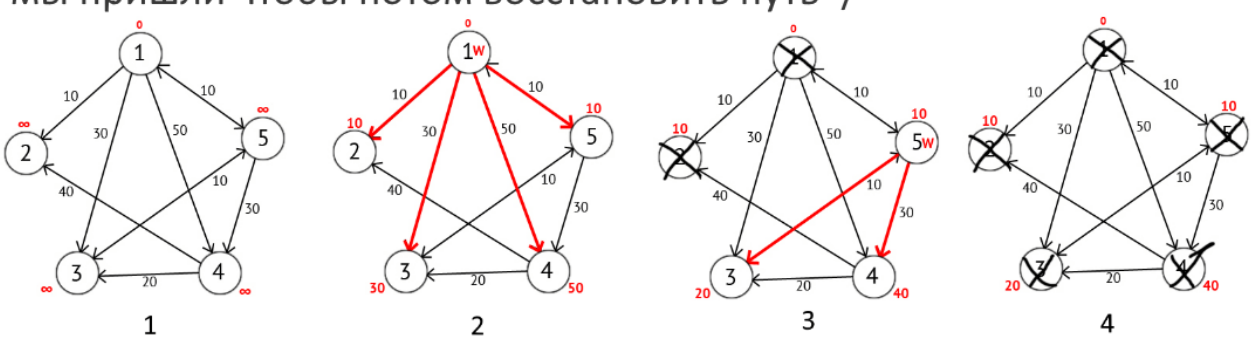
Для всех вершин в которые есть прямой путь из выбранной:

Берем сумму метки выбранной вершины и длины ребра в текущую

Если сумма меньше метки текущей вершины, обновляем значение метки

Помечаем выбранную вершину как обработанную

/*Мы получили длины путей из исходной вершины в любую другую.
На этапе обновления метки мы можем также запоминать вершину
из которой мы пришли чтобы потом восстановить путь*/

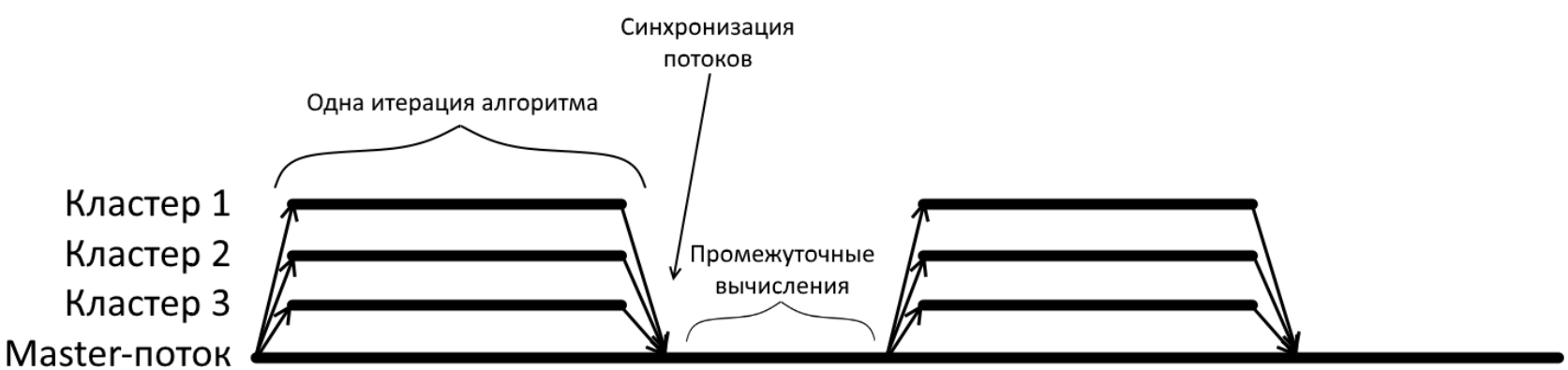


Дейкстра (Dijkstra) Параллельная реализация

Алгоритм не имеет полноценной параллельной версии, так что моя реализация предполагает разбиение графа на кластеры и их параллельную обработку, а также наличие master-потока для координации вычислений

Псевдокод (N-кол. вершин, T-кол. потоков помимо master):

Разбиваем граф на T частей //каждому кластеру достается примерно N/T вершин
Отмечаем исходную вершину как обработанную и добавляем к ответу
Пока не все вершины обработаны:
  Каждый кластер ищет еще не обработанную вершину, ближайшую к любой из обработанных вершин
  Находим ближайшую вершину из найденных кластерами, отмечаем и добавляем к ответу



Борувка (Boruvka) Классическая версия

Псевдокод

```
//Алгоритм берет все вершины исходного графа и добавляет к ним ребра
//Изначально, все все вершины являются отдельными компонентами связности
//Под объединением компонент понимается добавление соединяющего  их ребра
Пока ребер в ответе меньше чем вершин в графе -1:
   Берем любую компоненту
   Ищем кратчайшее ребро до любой другой компоненты
   Объединяем эти компоненты и добавляем ребро к ответу
```

Борувка (Boruvka) Параллельная реализация

Алгоритм имеет полноценную параллельную версию, все потоки равноправны

Псевдокот:

//Алгоритм берет все вершины исходного графа и добавляет к ним ребра
//Изначально, все все вершины являются отдельными компонентами связности
//Под объединением компонент понимается добавление соединяющего их ребра
Пока ребер в ответе меньше чем вершин в графе -1:
  Берем любую компоненту, которая в данный момент не обрабатывается другим потоком
  Ищем кратчайшее ребро до любой другой компоненты
  ПЫТАЕМСЯ объединить эти компоненты /*эта операция гарантирует корректную работу алгоритма при многопоточном выполнении, т.к. если другой поток уже объединил эти компоненты, то операция вернет false и поиск начнется заново*/
  Если объединение прошло успешно, добавляем найденное ребро к ответу

Поток 1
Поток 2
Поток 3

(Все потоки работают независимо друг от друга)

Заключение

В ходе работы, я написал собственные многопоточные реализации двух известных алгоритмов, а заодно и кучу различных структур и методов для работы с графами.
Все работает быстро и эффективно, раотой я доволен.
Вот примерная структура написанной библиотеки:

yaaz.dmex

Graph - класс графа,
имеет методы для итерации
по вершинам, ребрам, разбиения
на связные подграфы, итд
Graph.Vertex - класс вершины
Graph.[Named/Indexed/Point]Vertex -
различные расширения стандартного
класса вершины
Graph.Edge - класс ребра

yaaz.dmex.boruvka

Boruvka - класс для работы с алгоритмом

yaaz.dmex.dijkstra

Dijkstra - класс для работы с алгоритмом
PathNode - класс представляющий
результат работы алгоритма

yaaz.dmex.util

Пакет с различными классами
и методами для реализации
алгоритмов

(deafult package)

Main - класс с точкой входа
И кодом программы для
демонстрации

Исходный код можно посмотреть тут: https://bitbucket.org/YaaZ/dmex/src
Собранная программа: https://bitbucket.org/YaaZ/dmex/downloads/DMEX.jar

Нужна Java 8