Содержание

1 Введение

Задача обработки и анализа больших данных стоит повсеместно и возникает во множестве областей, как то известные поисковики, провайдеры, социальные сети, сотовые операторы, такие области науки как биоинформатика (обработка генов и генных сетей) и многие другие. Объёмы обрабатываемых данных от десятков гигабайт до террабайт и выше в зависимости от области. Такие данные зачастую представимы в виде графов, где вся необходимая информация хранится в вершинах и рёбрах или дугах графа и анализ таких данных сводится к классическим алгоритмам на графах. Одним из таких алгоритмов является алгоритм построение минимального остовного дерева.

В данной работе рассматривается задача построения минимального остовного дерева (minimum spanning tree — MST) на мультипроцессоре с ccNUMA архитектурой. Задача построения MST формулируется следующим образом: дан взвешенный, неориентированный граф и требуется найти остовное дерево (максимальный по включению рёбер подграф, не имеющий циклов), в котором сумма весов рёбер будет минимальна. В случае, когда исходный граф связный, в итоге будет построено остовное дерево, если же в исходном графе несколько компонент, то результатом будет лес.

Минимальные остовные деревья имеют широкое практическое применение. Они используются при построении различных сетей: коммуникационных линий, компьютерных сетей, транспортных сетей. Так же они используются в кластеризации, сегментации при обработке изображений, распознавании рукописного ввода.

2 Обзор алгоритмов построения MST

Существует несколько различных последовательных алгоритмов построения MST, наиболее распространёнными из которых являются алгоритмы Крускала, Прима и Борувки [1]. Алгоритм Крускала состоит из шагов (на каждом шаге добавляется одно новое ребро в текущий остов), каждый следующий из которых зависит от результата предыдущего, что делает распараллеливание данного алгоритма неэффективным. В алгоритме Прима шаги так же связаны между собой, а распараллеливание одного шага не даст большого выигрыша в силу большого количества шагов и малой вычислительной сложности одного шага. Однако, существует параллельный алгоритм построения MST, являющийся модификацией алгоритма Прима. Алгоритм Борувки в своём оригинальном виде является наиболее пригодным для распараллеливания. Далее опишем алгоритм Борувки и модифицированный алгоритм Прима.

2.1 Алгоритм Борувки

Изначально каждая вершина графа считается отдельной компонентой, затем компоненты объединяются до тех пор, пока не останется ровно одна. Объединение компонент происходит итеративно: на каждой итерации для каждой компоненты просматривается список всех инцидентных ей рёбер и выбирается минимальное из них, затем компоненты объединяются по найденным рёбрам.

За счёт того, что основная часть алгоритма – обход всех рёбер, который может быть разбит на независимые части и выполнятся параллельно, данный алгоритм в перспективе может дать хорошую масштабируемость. В статьях [2, 3, 4] представлены различные варианты параллельных реализаций алгоритма Борувки. Разные реализации используют разные структуры данных для хранения рёбер (хранение рёбер одним большим списком или хранение для каждой вершины списка инцидентных ей рёбер) и разные способы их объединения (слияние и копирование списков, объединение списков за константное время и т. д.).

2.2 Модифицированный алгоритм Прима

В статье [3] представлен параллельный алгоритм построения MST на основе последовательного алгоритма Прима. Суть данного алгоритма в том, что каждый поток выбирает случайную вершину в графе и начинает “растить” дерево из выбранной вершины, пока не попытается присоединить вершину, уже принадлежащую другому дереву, после чего два дерева встретившихся потоков объединяются, один из них продолжает “растить” объединённое дерево, а второй заново выбирает вершину графа. К преимуществам данного подхода относится отсутствие барьерной синхронизации, которая требуется в алгоритме Борувки после каждой итерации и, в зависимости от реализации, между разными шагами одной итерации.

3 Разработка параллельного алгоритма

В данных работах [2, 3, 4] результаты производительности показаны либо для старых архитектур [2, 4], либо масштабируемость была ограничена шестью одноядерными процессорами (SMP UMA, UltraSPARC II) [3] .

Настоящая работа направлена на адаптацию существующих и реализацию новых алгоритмов, ориентированных на высокую эффективность на современных вычислителях с общей памятью с ccNUMA архитектурой.

3.1 Алгоритм Борувки, использующий список рёбер

Первая параллельная реализация построения MST основана на алгоритме Борувки. Изначально множество вершин графа разделяется между потоками равномерно по количеству инцидентных рёбер. Далее на каждой итерации выполняются следующие шаги:

1. *Минимальное инцидентное ребро (1).* Каждый поток просматривает своё подмножество рёбер и для каждой компоненты находит инцидентное ребро минимального веса. В результате каждый поток будет иметь массив, содержащий минимальное ребро для каждой компоненты среди просмотренного подмножества.

2. *Минимальное инцидентное ребро (2).* Происходит редукция: для каждой компоненты находится ребро минимального веса по всем потокам. Таким образом, для каждой компоненты определяется минимальное ребро уже по всему графу.

3. *Объединение деревьев (1).* Для каждой компоненты определяется её номер на следующей итерации. Так же решается проблема с возможными циклами (когда более двух компонент по кругу выберут следующую в качестве ближайшей), которые возможны в алгоритме Борувки при наличии рёбер одинакового веса.

4. *Объединение деревьев (2).* Осуществляется перенумерация компонент: для каждой вершины вычисляется номер компоненты, в которую она входит, используя параллельный алгоритм Pointer Jumping [5].

Каждый шаг исполняется параллельно всеми потоками, однако после каждого шага требуется барьерная синхронизация всех потоков. Далее были сделаны следующие оптимизации:

- Сортировка рёбер по весу и удаление петель, за счёт чего в среднем сокращается количество рёбер, которые необходимо просмотреть на первом шаге.

- Выделение и инициализация памяти с ориентацией на NUMA архитектуру.

- Иерархическая редукция по дереву на втором шаге с учётом NUMA архитектуры.

3.2 Алгоритм Борувки, использующий отсортированные списки смежности

Наибольшая вычислительная сложность первого алгоритма заключается в первом шаге, в котором необходимо обойти бо̀льшую часть рёбер графа, однако, основным препятствием для хорошей масштабируемости является то, что размер массива, редуцируемого на втором шаге, увеличивается с ростом количества потоков.

Вторая реализация алгоритма Борувки для хранения рёбер использует списки смежности. Изначально для каждой вершины сортируется список инцидентных ей рёбер по увеличению веса. Таким образом, получить самое лёгкое ребро, инцидентное вершине, возможно за $O(1)$, не выполняя проход по массиву. Далее выполняются следующие шаги, пока не останется одна компонента:

1. *Минимальное инцидентное ребро.* Для каждой компоненты берётся инцидентное ей ребро минимального веса. Как отмечено выше, это будет первое ребро из списка смежности.

2. *Объединение деревьев.* Объединение компонент, для каждой компоненты определяется её номер на следующей итерации. Аналогично шагам *Объединение деревьев 1-2* в первом алгоритме.

3. *Слияние списков.* Происходит объединение компонент путём слияния списка рёбер. Поскольку, списки уже отсортированы по возрастанию веса, то возможно слияние, при котором в результате так же будет получен отсортированный список, с сохранением линейного времени работы.

4. *Перенумерация.* Обход списков рёбер и перенумерация компонент: старые номера заменяются на новые, полученные на шаге *Объединения деревьев*.

В ходе анализа работы шага *Объединение деревьев* на тестируемых типах графов было выяснено, что количество объединяемых компонент в одну в большинстве случаев не превосходит трёх и это позволяет быстро и эффективно отсеивать петли на шаге *Слияния списков* за счёт сокращения нерегулярных обращений в память.

3.3 Алгоритм Борувки, использующий неотсортированные списки смежности

Поддержка списков смежности отсортированными при слиянии требует больших временных затрат и несмотря на получившийся выигрыш в первом шаге, общее время выполнение алгоритма увеличилось. В связи с этим было решено отказаться от поддержания списков отсортированными. Таким образом, поиск минимума на первом шаге осуществляется обходом всего списка, но объединение списков стало возможно простым копированием.

Общая схема алгоритма осталась прежней, однако, реализация шага поиска минимального ребра и слияния списков значительно изменилась:

1. *Минимальное инцидентное ребро.* Для каждой компоненты находится инцидентное ей ребро минимального веса простым линейным поиском.

2. *Объединение деревьев.* Объединение компонент, для каждой компоненты определяется её номер на следующей итерации. Аналогично шагам *Объединение деревьев 1-2* в первом алгоритме.

3. *Слияние списков.* Происходит объединение компонент путём объединения списка рёбер.

4. *Перенумерация.* Обход списков рёбер и перенумерация компонент: старые номера заменяются на новые, полученные на шаге *Объединения деревьев.*

3.4 Алгоритм Борувки, использующий массив рёбер

Поскольку основная проблема описанных ранее подходов заключается в долго копировании данных на каждой итерации, то был разработан подход, позволяющий избежать копирования данных. Вместо объединения списков рёбер или вершин

Алгоритм одной итерации при таком подходе выглядит следующим образом:

1. *Минимальное инцидентное ребро.* Каждый поток просматривает своё подмножество рёбер и для каждой компоненты находит инцидентное ребро минимального веса. В результате каждый поток будет иметь массив, содержащий минимальное ребро для каждой компоненты среди просмотренного подмножества.

2. *Редукция.* Происходит редукция: для каждой компоненты находится ребро минимального веса по всем потокам. Таким образом, для каждой компоненты определяется минимальное ребро уже по всему графу.

3. *Объединение деревьев.* Для каждой компоненты определяется её номер на следующей итерации. Так же решается проблема с возможными циклами (когда более двух компонент по кругу выберут следующую в качестве ближайшей), которые возможны в алгоритме Борувки при наличии рёбер одинакового веса.

4. *Перенумерация.* Осуществляется перенумерация компонент: для каждой вершины вычисляется номер компоненты, в которую она входит, используя параллельный алгоритм Pointer Jumping [5].

4 Оптимизации EA реализации

4.1 Алгоритмические оптимизации

Предварительная сортировка рёбер. Алгоритм Борувки предполагает на каждой итерации просмотр всех рёбер графа, что и создаёт наибольшую вычислительную сложность шага ???. Первой проблемой такого подхода являются петли, который будут образовываться начиная с первой же итерации и которые затем будут каждый раз просматриваться. Удаление же петель повлечёт за собой перестроение графа, что, как было выявлено в ходе исследования, негативно сказывается на производительности. Второй проблемой является то, что одни и те же рёбра просматриваются многократно. ???

Решить данную проблему возможно предварительной сортировкой списков смежности каждой вершины по возрастанию веса. Затем для каждой вершины хранится индекс последнего просмотренного ребра: изначальной данный индекс указывает на первое ребро в списке, затем, по мере отбрасывания петель, он сдвигается и на следующей итерации поиск минимального ребра начинается не с начала списка, а с позиции, на которую указывает индекс. С другой стороны, если мы берём очередное ребро и его вес больше, чем текущий рекорд для компоненты, то поиск можно прервать, так как вес последующих рёбер данной вершины заведомо больше текущего.

Данная оптимизация хорошо масштабируется и на большом количестве потоков время выполнения невелико.

Переупорядочивание (перенумерация???) вершин. Вершины перенумеровывались в порядке обхода в ширину, предварительно отсортировав списки смежности каждой вершины по возрастанию степени вершины. Переупорядочивание вершин позволяет повысить локальность данных при последующих обходах графа [6] и, как следствие, ведёт к уменьшению кэш-промахов.

Отдельный цикл пропуска петель. В шаге a??? для каждой вершины ищется ребро минимального веса, ведущее в другую компоненту. С учётом оптимизации **предварительной сортировки рёбер** данный шаг выглядит следующим образом (упрощённо, конечно же):

Цикл может быть разбит и упрощён на более простой:

Написать, что петель отбрасывается очень много и потому данная модификация даёт улучшение производительности. Где статистика по петлям???

Балансировка больших вершин. Одной из особенностей RMAT-графов являет наличие вершин, степень которых значительно отличается от средней степени всех вершин графа. Например, в тестируемом RMAT-графе с 2^{20} вершинами (???) и со средней степенью 32, есть более ??? вершин степени выше 1000. Такие вершины создают дисбаланс на последних итерациях, когда просматривается большое количество рёбер и отбрасываются петли.

С целью устранения данного дисбаланса на стадии предобработки графа выделяются вершины степени более 1000 и равномерно распределяются между потоками. Значение степени 1000 было подобрано эмпирически и

```
Data: vertex v
myComp ← component[v];
foreach edgeId from edgesIdsBegin[v] to edgesIdsEnd[v] do
    destVertex ← edges[edgeId].destination;
    destComp ← component[destVertex];
    if destComp = myComp then
        continue
    end
    if edges[edgeId] < currentRecord[myComp] then
        currentRecord[myComp] ← edges[edgeId]
    end
    break;
end
```

Algorithm 1: Какой-то шаг до какой-то оптимизации

```
Data: vertex v
myComp ← component[v];
edgeId ← edgesIdsBegin[v];
while edgeId ≠ edgesIdsEnd[v] and comp = myComp do
    edgeId ← edgeId + 1;
end
if edges[edgeId] < currentRecord[myComp] then
    currentRecord[myComp] ← edges[edgeId]
end
```

Algorithm 2: Какой-то шаг после этой оптимизации

хорошо показало себя на практике.

4.2 Оптимизация редукции

Отсутствие редукции на первой итерации. Поскольку каждая компонента изначально представлена одной вершиной, то от редукции на первой итерации можно отказаться. Данная оптимизация выглядит простой и очевидной, однако, объём редуцируемого массива на первой итерации больше, чем на всех следующих, и данная модификация даёт существенное ускорение.

Перенумерация и сжатие массива компонент по ходу исполнения алгоритма. В начальной реализации при объединении компонент в качестве её номера выбирался один из номеров вершин, которые в неё входят. Таким образом, несмотря на сокращение числа компонент, диапазон используемых номеров не изменялся (минимальный и максимальный номер были приблизительно равны соответственно первой и последней вершине). Такой подход затруднял обход всех компонент на шаге редукции: требовалось либо обходить весь диапазон номеров и совершать лишнюю работу, просматривая "пустые" номера, не являющиеся компонентами, либо отдельно хранить массив, в которым хранить индексы всех "живых" на данный момент компонент, что усложняло реализацию и приводило к увеличению кэш-промахов из-за обхода массива с непостоянным шагом.

Явная перенумерация компонент на последовательные номера позволила решить обозначенную проблему. Более того, это позволило уменьшить объём используемой памяти: в начальной реализации для каждого потока создавался двумерный массив размера `количество_потоков`×`количество_вершин` На системах с большим количеством ядер, например, на Intel Xeon Phi, где используется до 240 потоков, объём такого массива заметно превосходит объём всех других данных и его сокращение в 2 раза приводит к значительному уменьшению потребляемой памяти (???).

Иерархическая редукция. Особенностью NUMA архитектуры являются высокие задержки при обращении в память другого узла NUMA. При стандартном подходе к редукции, когда для каждой компоненты последовательно обходятся все потоки и выбирается лучший результат, таких обращений при двух NUMA узлах будет как минимума, а при большем количестве узлов NUMA — ещё больше.

Данную проблему можно решить за счёт иерархического подхода к редукции. При таком подходе шаг редукции разбивается на два этапа: на первом происходит стандартная редукция внутри NUMA узла, а на втором — между узлами. Количество обращений к чужому узлу NUMA в этом случае уменьшается в количество раз, равное количеству используемых потоков в одном узле NUMA. Может, добавить красивую картинку(???)

Сообщения в редукции. В редуцируемом массиве много пустых элементов. Через сообщения можно уменьшить объём редуцируемых данных.

4.3 Программные/процессорные оптимизации ???

Программная предвыборка. Обход рёбер в общем случае неизбежно связан со случайными обращениями в память: невозможно отсортировать рёбра так, чтобы индексы обеих инцидентных вершин шли последовательно. Аппаратная предвыборка современных процессоров ориентирована только на последовательные обращения в память (точнее, на обход с постоянным, возможно отрицательным, шагом, размер которого не превосходит размер страницы памяти) и подобный обход рёбер вызывает простои, связанные с ожиданием данных.

Однако, адреса случайны обращений в память при таких обходах в большинстве случаев можно предсказать: обходя рёбра последовательно, нам известны индексы вершин для следующих итераций. Как раз в таком случае и помогает программная предвыборка, давая возможность заранее запросить нужные данные и сократить время простоя.

Другой особенностью аппаратной предвыборки является то, что данные не всегда загружаются в кэш первого уровня (например, на Intel Xeon Phi).

Сказать про двойную предвыборку ???

Выделение памяти с учётом NUMA архитектуры Как уже отмечалось ранее, обращение в память другого узла NUMA связано с высокими задержками. Выделение памяти на тех ядрах, где она используется, является более правильным на NUMA архитектуре.

В случае использования двумерных массивов, когда первая размерность соответствует номеру потока (например, массив в котором хранится текущий лучший результат для каждой компоненты), основной интерес представляет вторая размерность, поскольку к первой будет только одно обращение за итерацию, а ко второй — на каждой вершине. Таким образом, первая размерность выделяется на произвольном узле NUMA, а затем каждой поток выделяет необходимую ему память (написано криво???).

Когда одномерный массив одновременно использует несколько потоков (например, массив рёбер), то возникает необходимость выделить непрерывную область в памяти на разных узлах NUMA. Ядро Linux предоставляет разработчику возможность явно управлять выделением страниц памяти на NUMA системах с использованием libnuma. Однако, в реализациях использовалась особенность ядра Linux известная как memory overcommit: выделение страниц памяти происходит не в момент вызова функции malloc, а в момент первого обращения к странице. Таким образом, для разделения массива по узлам NUMA сначала с использованием функции malloc выделялся весь, далее, каждый поток инициализировал те страницы, которые он будет использовать. При таком подходе, конечно же, может возникнуть пересечение, когда два потока используют одну страницу, но размеры таких пересечений невелики по сравнению с размером всего массива.

Был также опробован подход с дублированием массива, который используют одновременно несколько узлов NUMA. Одним из примеров таких данных является массив, где для каждой вершины хранится индекс компоненты, в которую она входит на данный момент. Однако, из-за необходи-

мости после каждой итерации дополнительно обновлять массив на каждом узле NUMA данный подход себя не оправдал.

Аппаратные потоки процессора. Современные процессоры позволяют на одном ядре исполнять (???) одновременно несколько аппаратных потоков. Так в десктопных (домашние???) и серверных процессорах Intel и AMD данные технологии называются соответственно Hyper-Threading и Modules и позволяют запускать по 2 аппаратных потока на ядро, а ускоритель Intel Xeon Phi на данный момент имеет 4 аппаратных потока на каждое ядро. Использование такого вида многозадачности позволяет не только более эффективно использовать processor units, но и скрыть задержки в память (за счёт переключение на другой поток???).

Проблема использования аппаратных потоков для EA-реализации заключается в увеличении объёма шага редукции. В начальных реализациях использование Hyper-Threading давало небольшое (какое???) ускорение на малом количестве потоков, но ухудшало время на 16 потоках. После применения описанных выше оптимизаций редукции Hyper-Threading стал давать выигрыш в среднем 5%.

Linux huge pages. Увеличение объёма страниц памяти ⇒ уменьшение количества страниц ⇒ меньше промахов в TLB кэш. К сожалению, на данный момент хоть и huge pages включены в ядре Linux, однако, настройки системы по умолчанию не позволяют их использовать без дополнительных действий со стороны администратора системы, потому данная оптимизация не была протестирована на Intel Xeon.

Чуть лучше дела обстоят на ускорителе Intel Xeon Phi, где huge pages разрешены “из коробки”. Использование huge pages на данном ускорителе дало такой-то выигрыш.

4.4 Результаты

Тут должны быть графики.

5 Заключение

На данный момент исследованы последовательные алгоритмы построения остовных деревьев и выбраны из них те, которые или модификации которых могут дать хорошую масштабируемость при распараллеливании. Так же изучены статьи [2, 3, 4] по распараллеливанию данных алгоритмов.

Были написаны последовательные реализации алгоритмов для дальнейшей проверки корректности получаемых результатов. Затем был написан и распараллелен алгоритм Борувки в нескольких реализациях, использующих разные структуры данных для хранения графа.

В дальнейших планах есть как оптимизация существующих алгоритмов, так и реализация новых.

В уже реализованных алгоритмах ближайшей целью является использование таких подходов как параллельные алгоритмы поиска компонент связности в тех шагах, где происходит объединение вершин одной компоненты.

Далее возможно использование других структур данных в алгоритме Борувки. Рассматриваются такие структуры, как списки отсортированных списков смежностей, являющиеся промежуточным вариантом между поддержанием списков смежностей отсортированными на каждой итерации и полным отказом от сортировки списков, и деревья или кучи, позволяющие при дополнительных условиях производить их слияние за время, пропорциональное их высоте.

Так же планируется не только модификация алгоритма Борувки, но и реализация модифицированного алгоритмы Примы.

Список литературы

[1] Т. Кормен. *Алгоритмы: построение и анализ.* Вильямс, 2005.

[2] Silvia Gotz Frank Dehne. Practical parallel algorithms for minimum spanning trees. *http://people.scs.carleton.ca/~dehne/publications/2-47.pdf*, 1998.

[3] Guojing Cong David A. Bader. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. *http://www.cc.gatech.edu/~bader/papers/MST-JPDC.pdf*, 2006.

[4] Anne Condon Sun Chung. Parallel implementation of boruvka minimum spanning tree algorithm. *http://minds.wisconsin.edu/bitstream/handle/1793/60020/TR1297.pdf*, 1996.

[5] Bruce M. Maggs Guy E. Blelloch. Parallel algorithms. *http://homes.cs.washington.edu/~arvind/cs424/readings/BlellochM96b.pdf*, 2004.

[6] J. McKee Elizabeth Cuthill. Reducing the bandwidth of sparse symmetric matrices. *ACM '69 Proceedings of the 1969 24th national conference*, pages 157–172, 1969.