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 построения минимального остовного дерева.

В данной работе рассматривается задача построения минимального остовного дерева (minimum spanning tree — MST) на мультипроцессоре с ccNUMA архитектурой. В теории графов остовным деревом называется ацикличный связный подграф данного связного неориентированного графа, в который входят все его вершины. Минимальными остовным деревом называется остовное дерево во взвешенном графе с минимальной суммой весов рёбер. В случае, когда данный граф не является связным, для каждой компоненты будет построено своё остовное дерево и полученные деревья называются остовным лесом.

Минимальные остовные деревья имеют широкое практическое применение. Они используются при построении различных сетей: коммуникационных линий, компьютерных сетей, транспортных сетей [1]. Так же они используются в задачах кластеризации [2], сегментации при обработке изображений [3], распознавании рукописного ввода [4]. Минимальные остовные деревья используются как приближённое решение переборной задачи: например, задачи Штейнера — построение кратчайшей сети, соединяющей заданный конечный набор точек на плоскости [5].

2 Обзор алгоритмов построения MST

Существуют различные последовательные алгоритмы построения MST, наиболее распространёнными из которых являются алгоритмы Крускала, Прима и Борувки [6]. Не каждый из алгоритмов в его оригинальном варианте возможно эффективно распараллелить, однако, для некоторых из них существуют модификации, за счёт которых использование нескольких ядер одного узла кластера и даже использование нескольких узлов кластера может оказаться оправданным [7, 8].

Алгоритм Крускала строит остовное дерево в два этапа: сначала все рёбра графа сортируются в порядке неубывания веса, затем рёбра последовательно обходятся и в остовное дерево добавляются те из них, которые не создают цикла.

Сортировка рёбер в общем случае может быть выполнена за время $O(E \cdot \log(E))$, проверка образования цикла для каждого ребра может быть реализована с помощью системы непересекающихся множеств и выполняться за время $O(\alpha(E, V))$. Итоговая оценка времени работы получается $O(E \cdot \log(E) + \alpha(E, V))$.

Алгоритм Прима начинает построение остовного дерева с произвольно выбранной вершины. Затем, пока дерево не будет построено полностью, выбирается ребро минимального веса, соединяющее вершину уже построенного дерева с вершиной не из дерева и данное ребро добавляется в строящееся остовное дерево.

Число шагов в алгоритме Прима равно числу вершин графе. Время поиска ребра минимального веса на каждом шаге зависит от выбранной структуры данных: при использовании матрицы смежности для поиска минимального ребра потребуется просмотреть всю строку матрицы за линейное время, а при использовании кучи добавление и удаление каждого ребра будет происходить за логарифмическое время. Таким образом, алгоритм Прима возможно реализовать со временем работы $O(V^2)$, что является лучшей оценки для полных графов, либо со временем $O(E \log(V))$, что более предпочтительно для разреженных графов.

Алгоритм Борувки. Это итерационный алгоритм: изначально каждая вершина графа считается отдельной компонентой, затем на каждой итерации для каждой компоненты находится инцидентное ей ребро минимального веса и по найденным рёбрам компоненты объединяются. Алгоритм работает до тех пор, пока не останется одна компонента.

На каждой итерации алгоритма множество компонент уменьшается не менее, чем в два раза, следовательно количество итераций можно оценить сверху как $\lceil \log_2(V) \rceil$. На каждой итерации в худшем случае необходимо просмотреть все рёбра, при этом процедура обработки одного ребра достаточно простая (определение каким компонентам принадлежат инцидентные вершины и, возможно, обновление рекорда для данных компонент). Итоговая временная сложность получается $O(E \cdot \log(V))$.

2.1 Параллельные алгоритмы

С точки зрения параллелизма про алгоритмы Крускала и Прима можно сказать следующее:

- они состоят из большого количества шагов, которые зависимы между собой и не могут исполняться параллельно
- вычислительная сложность одного шага достаточно маленькая и её распараллеливание будет неэффективно

Это делает распараллеливание данных алгоритмов в оригинальном варианте бесперспективным.

Исключением является реализация алгоритма Прима для плотных графов с использованием массивов вместо куч: в этом случае сложность одного шага составляет $O(V)$ и её возможно распараллелить, разделив вершины между потоками. Однако, не для плотных графов такой подход асимптотически хуже других алгоритмов и при распараллеливании даже на большое число вычислителей будет медленнее последовательных реализаций других алгоритмов.

В то же время алгоритм Борувки обладает противоположными свойствами:

- количество итераций небольшое
- вычислительная сложность одной итерации большая и может быть распределена между потоками

Это делает алгоритм Борувки наиболее пригодным для распараллеливания и в перспективе может дать хорошую масштабируемость.

В статьях [9, 7, 10] 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.).

В статье [7] 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. Другим преимуществом перед алгоритмом Борувки является то, что каждое ребро будет просмотрено ровно один раз.

3 Варианты алгоритма Борувки

В данных работах [9, 7, 10] результаты производительности показаны либо для старых архитектур [9, 10], либо масштабируемость была ограничена шестью одноядерными процессорами (SMP UMA, UltraSPARC II) [7].

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

В частности, далее будут представлены результаты тестирования описанных реализаций на двухсокетной системе 2 × Intel Xeon CPU E5-2690 (8 ядер 2.9 GHz, 32KB L1, 256KB L2, 20MB L3).

3.1 Списки смежности (Adjacency Lists — AL)

Первая реализация алгоритма Борувки для хранения графа использует списки смежности для каждой компоненты. Изначально каждая компонента представлена одной вершиной и её список смежности совпадает со списком смежности вершины, затем после каждой итерации для каждой образовавшейся компоненты явно строится список всех рёбер, инцидентных данной компоненте. В качестве списков может быть использована любая структура данных, позволяющая последовательно обходить элементы. В данном подходе использовались динамически выделяемые массивы.

Изначально множество вершин графа разделяется между потоками равномерно по количеству инцидентных рёбер. Далее на каждой итерации алгоритма Борувки выполняются следующие шаги, пока не останется одна компонента или между несколькими оставшимися компонентами не будет рёбер:

1. *Минимальное инцидентное ребро.* Каждый поток для каждой своей компоненты находит инцидентное ей ребро минимального веса. Это делается обходом всех инцидентных компоненте рёбер.

2. *Объединение деревьев.* Компоненты объединяются по найденным рёбрам: выделяются образовавшиеся компоненты, для каждой такой компоненты определяется номер на следующей итерации. Так же решается проблема с возможными циклами (когда более двух компонент по кругу выберут следующую компоненту в качестве ближайшей), которые возможны в алгоритме Борувки при наличии в графе рёбер одинакового веса.

3. *Перенумерация компонент.* Осуществляется перенумерация компонент: для каждой вершины вычисляется номер компоненты, в которую она входит, используя параллельный алгоритм Pointer Jumping [11].

4. *Слияние списков.* Происходит объединение компонент путём слияния списка рёбер. В модификации с неотсортированными списками рёбер слияние делается последовательным копированием одно списка в другой. При использовании отсортированных по весу списков вместо копирования и последующей сортировки рёбер происходит слияние с использованием приёма, который используется в сортировки слиянием: из голов копируемых списков выбирается минимальные элемент, извлекается и добавляется в создаваемый список.

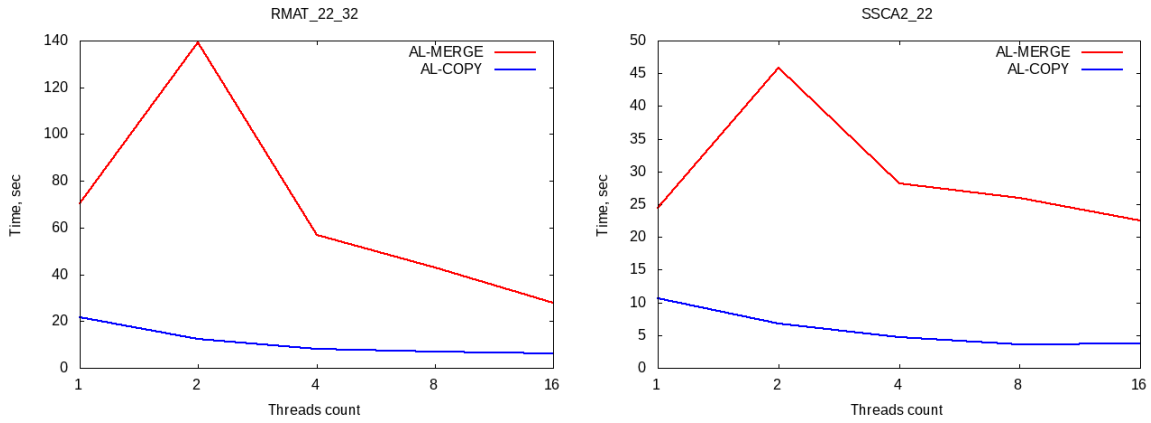
Были опробованы два подхода относительно сортированности списков смежности:

- **Отсортированные по весу списки смежности**. Отсортированные списки позволяют осуществлять шаг *минимальное инцидентное ребро* за константное время,

просмотрев лишь голову списка, однако поддержание таких списков сортированными на шаге *слияния списков* требует дополнительного времени.

- **Неотсортированные списки смежности**. Когда поддержание списков отсортированными не требуется, их объединение возможно осуществлять простым копированием памяти, но поиск по неотсортированному списку возможен только за линейное время.

На практике подход с сортировкой списков по весу оказался заметно медленнее. цирфы??? ниже



В данном подходе имеется две существенные проблемы. Первая из них заключается в том, что на шаге *слияния списков* происходит копирования большого объёма данных и этот шаг исполняется долго. Вторая проблема лежит в том, что при распределении компонент между потоками на шагах *минимальное инцидентное ребро* и *слияния списков* происходит дисбаланс нагрузки на последних итерациях, когда количество оставшихся компонент становится меньше числа потоков — в этом случае некоторые потоки просто простаивают без работы, что негативно сказывается на масштабируемости алгоритма.

3.2 Списки вершин (Vertexes lists — VL)

В качестве решения первой проблемы — большого объёма копируемых данных после каждой итерации — было решено (???) заменить списки смежности каждой компоненты на список входящих в неё вершин. Во-первых, такая модификация минимально влияет на другие шаги алгоритма. Во-вторых, на шаге *слияния списков* это позволяет в несколько раз сократить размер объединяемых списков.

Общая схема алгоритма осталась прежней со следующими изменениями:

1. На шаге *минимальное инцидентное ребро* теперь для обхода рёбер требуется обойти не список рёбер, а список входящих в компоненту вершин и уже для каждой вершины обойти список инцидентных ей рёбер.

2. На шаге *слияние списков* теперь копируются не списки рёбер, а списки вершин, что значительно сокращает объём перемещаемых данных.

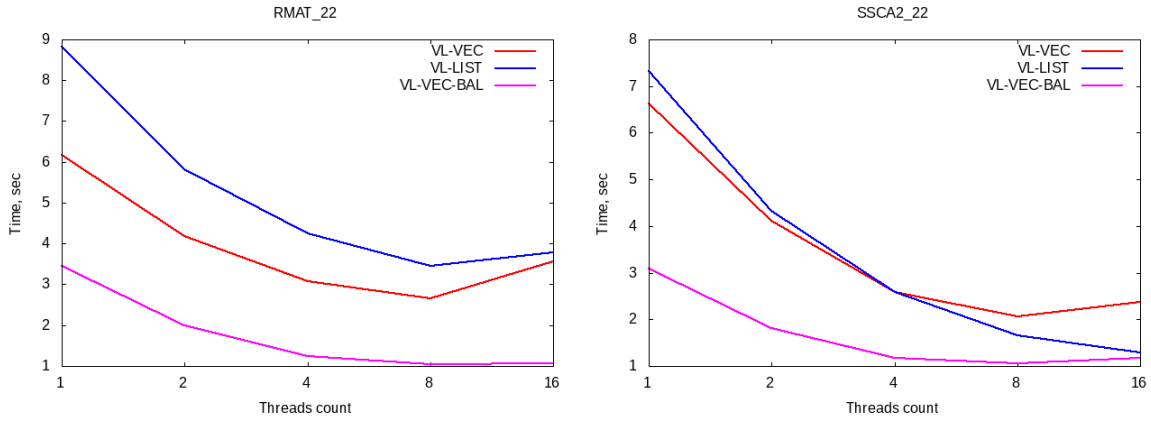
Для хранения списка вершин были опробованы две структуры данных:

- **Односвязные списки**. Их преимущество заключается в возможности объединения за константное время, однако, обход таких списков связан со случайными обращениями в память.

- **Динамические массивы**. Объединение массивов возможно только за линейное время путём копирования одного из них в другой (на практике используется копирование меньшего массива в больший). Но во время обхода массива хорошо работает аппаратная предвыборка, что позволяет в значительной степени избежать кэш-промахов.

Данные модификации показывали примерно одинаковую производительность, однако, в среднем реализация с использованием динамических массивов оказалась на 5-10% быстрее.

В качестве решения второй проблемы — дисбаланс на последних итерациях — шаг *минимальное инцидентное ребро* был разделён на два: сначала выделяются компонент с небольшим количеством входящих в них вершин и обрабатываются как и раньше. Затем остаётся несколько компоненты, состоящие из большого числа вершин, и работа по таким компонентам уже разделяется между потоками: каждый поток обрабатывает свою часть вершин, находит среди них минимальное ребро и в конце по всеми потокам находится. Данная оптимизация улучшила производительность алгоритма и позволила получить ускорение на 2 сокетах.



3.3 Массив рёбер (Edges Array — EA)

Поскольку за счёт предыдущей модификации полностью решить проблему копирования больших объёмов данных не удалось, то следующий алгоритм разрабатывался с целью полностью избежать перемещения списков рёбер или вершин после каждой итерации. Если ранее использовался подход, когда для каждой компонента хранилась информация, позволявшая восстановить список инцидентных рёбер, то теперь будет хранится информация для каждой вершины о том, какой компоненте она принадлежит.

Изначально все вершины распределяются между потоками равномерно по количеству рёбер. Далее на каждой итерации алгоритма Борувки выполняются следующие шаги:

1. *Минимальное инцидентное ребро*. Каждый поток обходит все свои вершины и находит минимально ребро для всех компонент, которые представлены в данном потоке хотя бы одной вершиной. В результате каждый поток будет иметь массив, содержащий минимальное ребро для каждой компоненты среди просмотренного подмножества.

2. *Редукция*. Происходит редукция полученного на предыдущем шаге массива: 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 [11].

Основная проблема данного подхода заключается в шаге *редукции*: объём редуцируемого массива растёт линейно с ростом числа потоков, таким образом, среднее время выполнения данного шага почти не уменьшается с ростом количества потоков. Однако, даже с описанной проблемой подход с уменьшением объёма копируемых данных себя оправдал и данная реализации показала лучшие результаты производительности среди описанных алгоритмов. Далее будут подробно рассмотрены оптимизации, применённые к данному подходу хранения графа.

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 первого шага. Объём "лишней" работы при этом достаточно велик: во-первых, постоянно будет просматриваться большое количество петель, которых с каждой итерацией становится всё больше и больше, во-вторых, почти все рёбра, не являющиеся петлями, будут отбрасываться *(в значении пропускаться до следующей итерации???)*, как более тяжёлые.

Проблему с петлями возможно попытаться решить их удалением, однако, это повлечёт за собой частичное перестроение *структуры?внутреннего представления???* графа, что, как было выявлено в ходе исследования, негативно сказывается на производительности.

Решить обе обозначенные проблемы возможно за счёт предварительной сортировки списков смежности каждой вершины по возрастанию веса рёбер. 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 рекорда.

Данная оптимизация хорошо масштабируется и на большом количестве потоков время выполнения невелико. В среднем производительность от данной оптимизации увеличивается на 20%.

Перенумерация вершин. В работе [12] предлагается алгоритм перенумерации вершин, который повышает локальность данных при последующих обходах графа, за счёт чего уменьшается количество кэш-промахов. Предложенный алгоритм перенумеровывает вершины в порядке обхода в ширину с предварительной отсортировкой списков смежности каждой вершины по возрастанию степени вершины, в которую ведёт ребро.

Балансировка больших вершин. Одной из особенностей RMAT-графов является 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 1351 вершина 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 хорошо показало себя на практике.

В среднем оптимизации **перенумерация вершин** и **балансировки больших вершин** дают увеличение производительности в 11%.

Отдельный цикл пропуска петель. Шаг *поиск минимального ребра* 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???

В среднем данная оптимизация ведёт к увеличению производительности на 10%.

```
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

4.2 Оптимизация редукции

Поскольку основные проблемы масштабируемости лежат именно в *шаге редукции*, то было сделано несколько оптимизаций, непосредственно затрагивающих данный шаг. Приведённые оценки увеличения производительности получены при запуске 32 потоков на всех 16 ядрах процессоров (исполнялось по 2 потока на одно ядро, поскольку в процессере была включена технология Hyper Threading, использование которой описано далее) и показывают ускорение не одного только шага *редукции*, а время исполнение всего алгоритма.

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 ускорение в 20%.

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 поддерживать 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 перед каждой итерацией на последовательные номера позволила решить обозначенную проблему. В среднем производительность увеличилась на 20%.

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 минимум половина, а при большем количестве узлов NUMA — ещё больше. (тут можно указать достаточно точную оценку $\frac{\texttt{число NUMA узлов - 1}}{\texttt{число NUMA узлов}}$, но стоит ли???)

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. В результате иерархическая редукция повысила производительность на 5%. Может, добавить красивую картинку(???)

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

4.3 Архитектурные 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 случайных 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 для будущих вершин (например, какой компоненте они принадлежат) и сократить время ожидания.

В среднем программная предвыборка позволила повысить производительность на 30%. Сказать про то, что стадии типа PJ и того можно ускорить в несколько раз.

Иногда, в одном цикле целесообразно использование программной предвыборки для двух массивов с разным шагом. В коде:

for $vertexId$ *from* 1 *to* $vertexCount$ **do**
  $edgeId \leftarrow startEdge[vertexId]$;
  $destComp \leftarrow component[edges[startEdge].dest]$;
  ...
end

Algorithm 3: возможно, данный заголовок стоит выпилить

обращение к массиву *component* есть возможность

Однако, здесь адрес второй предвыборки зависит от случайного обращения к элементу $edge[startEdge[vertexId + prefetchStep]$ и более эффективным будет заранее загрузить в кэш данный элемент:

for *vertexId from* 1 *to vertexCount* **do**
  $prefetch(\&component[edge[startEdge[vertexId + prefetchStep]].dest])$;
  $edgeId \leftarrow startEdge[vertexId]$;
  $destComp \leftarrow component[edges[startEdge].dest]$;
  ...
end

Algorithm 4: возможно, данный заголовок стоит выпилить

for *vertexId from* 1 *to vertexCount* **do**
  $prefetch(\&edge[startEdge[vertexId + prefetchStep \times 2]])$;
  $prefetch(\&component[edge[startEdge[vertexId + prefetchStep].dest]])$;
  $edgeId \leftarrow startEdge[vertexId]$;
  $destComp \leftarrow component[edges[startEdge].dest]$;
  ...
end

Algorithm 5: возможно, данный заголовок стоит выпилить

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 и, возможно, стоит улучшить???).

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 необходимости после каждой итерации дополнительно обновлять массив на каждом узле NUMA данный подход себя не оправдал и производительность в лучшем случае не уменьшалась.

Аппаратные потоки процессора. Современные процессоры позволяют на одном ядре исполнять одновременно несколько аппаратных потоков. Так в 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 процессорные элементы, но и скрыть задержки в память за счёт переключения на другой поток.

Проблема использования аппаратных потоков для EA-реализации заключается в увеличении объёма данных на шаге *редукции*. В начальных реализациях использование Hyper-Threading давало небольшое 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.

5 Результаты

5.1 Окружение

Описание характеристик используемых машин (2xIntelXeon, Intel Xeon Phi), описание используемых графов (rmat, ssca2, возможно решётку и случайные графы), формат входных данных (матрица в формаате CSR), как измерялось время (rdtsc, сколько замеров, минимальный результат).

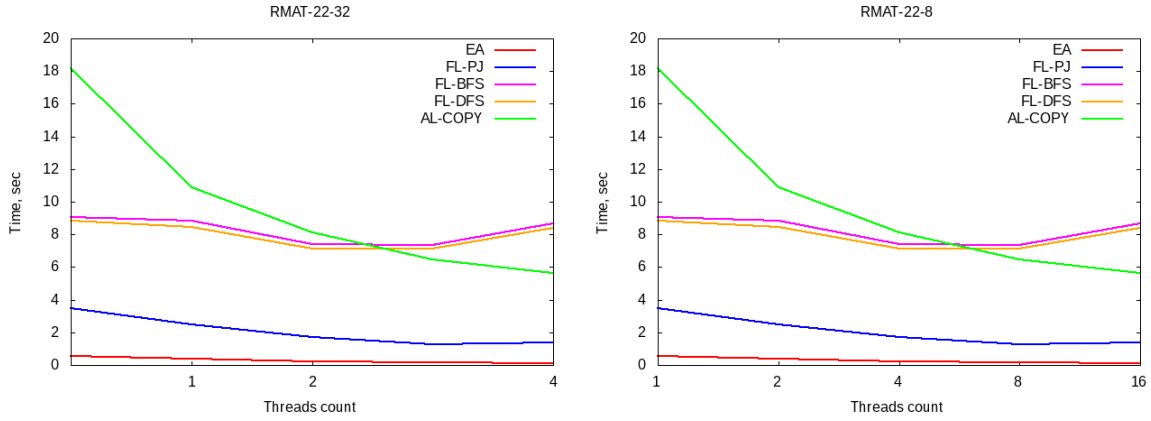
Тестирование проводилось на следующих графах:

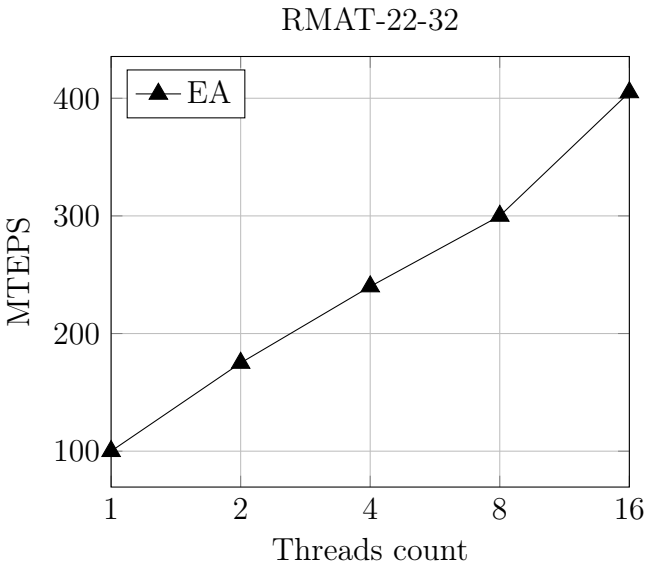
- **RMAT-графы**. Структура RMAT-графа напоминает структуру социальной сети.
- **SSCA2-графы**. Данные графы представляют из себя что??? Состоят из клик, между которые случайно набросаны рёбера.
- **Двумерные и трёхмерные решётки**.
- **Случайные графы (???)**.

В качестве формата хранения графа использовался сжатый формат хранения матриц CSR.

5.2 Непосредственно графики

Тут должны быть графики и числа. Сравнение всех реализаций на RMAT-22, SSCA2-22 (возможно, решётки и случайные графы). Сравнение лучшей реализации на нескольких RMAT и SSCA2. Возможно, графики для Intel Xeon Phi.




RMAT-22-32
EA
400
300
200
100
MTEPS
1
2
4
8
16
Threads count

6 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 [9, 7, 10] по распараллеливанию данных алгоритмов.

Были получены 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] Raúl Tapia, Ernesto; Rojas. *Recognition of On-line Handwritten Mathematical Expressions Using a Minimum Spanning Tree Construction and Symbol Dominance*. 2004.

[2] B.; Keil M.; Yao F. Asano, T.; Bhattacharya. *Clustering algorithms based on minimum and maximum spanning trees*. Fourth Annual Symposium on Computational Geometry, 1988.

[3] J.; Michel O. Ma; Hero, A.; Gorman. *Image registration with minimum spanning tree algorithm*. 2000.

[4] Raúl Tapia, Ernesto; Rojas. *Recognition of On-line Handwritten Mathematical Expressions Using a Minimum Spanning Tree Construction and Symbol Dominance*. 2004.

[5] Ткачев С. Б. Белоусов А. И. *Дискретная математика*. МГТУ, 2006.

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

[7] 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.

[8] Srdjan Skrbic Vladimir Loncar and Antun Balaz. Parallelization of minimum spanning tree algorithms using distributed memory architectures. 2014.

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

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

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

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