Параллельный алгоритм поиска минимального остовного дерева в графе на суперкомпьютере с сетью «Ангара»

А.В. Мазеев

АО «НИЦЭВТ»

В данной работе предложен параллельный алгоритм решения задачи поиска минимального остовного дерева в больших графах для вычислительных систем с распределенной памятью. Базовым алгоритмом для разработанного является алгоритм GHS. Разработка и исследование алгоритма производились на 36-узловом вычислительном кластере «Ангара-К1», с использованием языка программирования C++ и библиотеки передачи сообщений MPI.

Ключевые слова: графы, MST, суперкомпьютеры, сеть «Ангара», MPI

1. Введение

Пусть $G = (V, E)$ — связный взвешенный неориентированный граф. Остовное дерево — дерево в этом графе, которое содержит все вершины. Минимальное остовное дерево MST (англ. Minimum Spanning Tree) [1] — это остовное дерево, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов входящих в него ребер.

В данной работе рассматривается проблема поиска минимального остовного дерева в больших графах. Под большими подразумеваются графы, которые не могут быть представлены в оперативной памяти типичного узла распределенной вычислительной системы.

Задача поиска MST встречается во многих областях, например, в компьютерном зрении, биоинформатике, а также при проектировании различных сетей. В реальных задачах постоянно растут требования к объему обрабатываемых графов. Например, в биоинформатике для решения задачи кластеризации [2], которая может быть решена при помощи построения MST, необходимые для обработки графы могут занимать терабайты памяти и больше.

Существует большое количество алгоритмов [3], решающих задачу поиска MST; самые известные из них — алгоритмы Прима [4], Крускала [5] и Борувки [6]. Некоторые алгоритмы удобны для распараллеливания на общей памяти, существует множество таких реализаций, например [7–10].

Часть перечисленных алгоритмов адаптирована для реализации на распределенной памяти [13–16]. Среди перечисленных выше параллельных реализаций нет ни одной, масштабируемой хотя бы на несколько десятков ядер вычислительной системы с распределенной памятью. Существуют алгоритмы, специально созданные для распределенных систем, например, алгоритм GHS (англ. Gallager, Humblet, Spira) [11] и Awerbuch [12]. Найдена только одна работа [15], в которой описана реализация алгоритма GHS, однако в этой работе недостаточно данных для адекватного сравнения результатов.

В данной работе на основе алгоритма GHS разработан параллельный алгоритм поиска минимального остовного дерева для вычислительных систем с распределенной памятью, который позволяет обрабатывать большие графы и получать высокие показатели эффективности. Данная работа является развитием результатов, опубликованных в статье [17].

2. Описание алгоритма GHS

Алгоритм GHS выбран для исследования как базовый и наиболее простой. Этот алгоритм очень хорошо описывается в модели vertex-centric [18]. Идея алгоритма состоит в

следующем. Все вершины графа выполняют одну и ту же процедуру, которая заключается в отправке, приеме и обработке сообщений от смежных вершин. По каждому ребру графа сообщения передаются независимо в обоих направлениях, причем они не должны обгонять друг друга в рамках одного направления по ребру.

В каждый момент времени множество вершин графа представляется в виде объединения некоторого количества фрагментов, то есть непересекающихся множеств вершин. Изначально каждая вершина является фрагментом. Каждый фрагмент находит среди ребер, исходящих из него в другие фрагменты, ребро с минимальным весом (минимальное исходящее ребро). Далее по этим ребрам фрагменты объединяются. Ребра, по которым происходит объединение фрагментов, будут составлять минимальное остовное дерево, когда для связного графа останется ровно один фрагмент, включающий в себя все вершины.

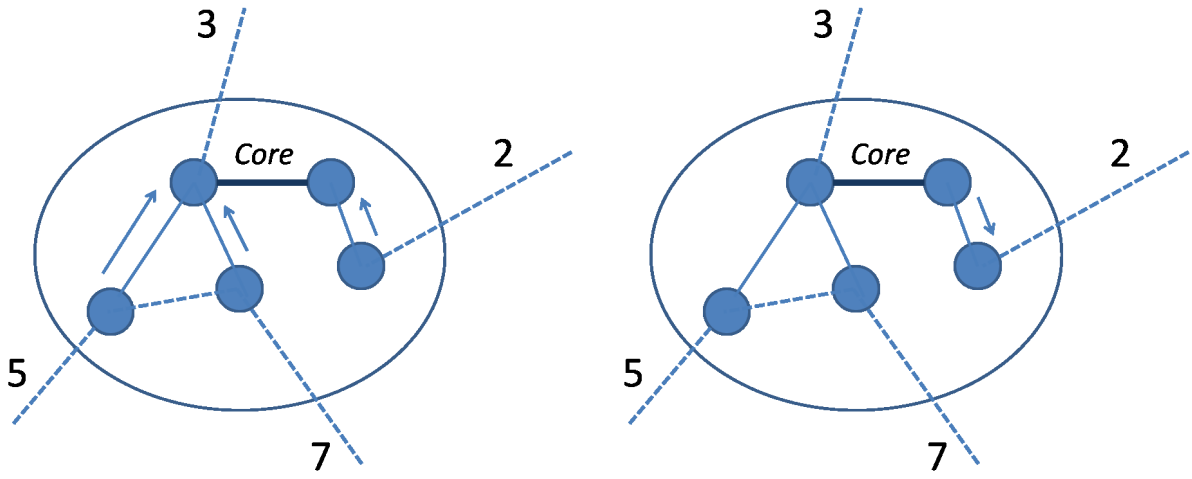
Рассмотрим работу алгоритма подробнее. Существует три возможных состояния вершины: *Sleeping* — начальное состояние, *Find* — когда вершина участвует в поиске минимального исходящего из фрагмента ребра, и *Found* — в остальных случаях. У каждого фрагмента есть переменная L, характеризующая его уровень. Изначально уровень каждого фрагмента равен 0. Два фрагмента с одинаковым уровнем L могут объединиться во фрагмент с уровнем $L + 1$. Фрагмент не может присоединиться к фрагменту с меньшим уровнем.

Опишем подробнее, как происходит поиск минимального исходящего ребра во фрагменте. В тривиальном случае, когда фрагмент состоит из одной вершины, и его уровень равен 0, вершина локально находит минимальное исходящее ребро, помечает его как часть минимального остовного дерева и отправляет сообщение *Connect* по этому ребру и переходит в состояние *Found*.

Теперь рассмотрим случай, когда уровень фрагмента больше 0. Предположим, что два фрагмента уровня $L-1$ объединились во фрагмент уровня L по одному исходящему ребру. Ребро, по которому объединились два фрагмента, становится ядром нового фрагмента. Вес ядра используется в качестве идентификатора фрагмента. Далее сообщение *Initiate* рассылается по всему фрагменту, начиная от вершин, смежных с ядром. Сообщение *Initiate* рассылается по фрагменту для того, чтобы все вершины получили новый уровень и идентификатор фрагмента, а также установили свое состояние в *Find*. Когда вершина получает сообщение *Initiate*, она начинает участвовать в поиске минимального исходящего ребра.

Каждое ребро графа может быть в одном из трех состояний: *Branch* — ребро принадлежит минимальному остовному дереву, *Rejected* — не принадлежит, *Basic* — в случае, если пока не известно, будет это ребро принадлежать минимальному остовному дереву или нет. Чтобы найти минимальное исходящее ребро, в некоторой вершине v поочередно перебираются, начиная с самого легкого, ребра, находящиеся в состоянии *Basic*. Для каждого такого ребра происходит проверка при помощи посылки по нему сообщений типа *Test*. Сообщение *Test* передает уровень и идентификатор фрагмента. Когда вершина u принимает сообщение *Test*, она сравнивает свой идентификатор фрагмента с принятым в сообщении. Если идентификаторы равны, тогда вершина u отправляет в ответ сообщение *Reject*, и после этого обе вершины устанавливают состояние ребра в *Reject*. В этом случае вершина v, которая отправила сообщение *Test*, продолжает поиск, анализируя следующее наилучшее ребро и так далее. Если идентификатор фрагмента в полученном сообщении *Test* отличен от идентификатора фрагмента принимающей вершины u, и если уровень фрагмента у принимающей вершины больше либо равен, чем в сообщении *Test*, тогда в ответ отправляется сообщение *Accept*. Состояние ребра у вершины v в этом случае меняется на *Branch*. Однако если уровень фрагмента вершины u меньше, чем в пришедшем сообщении, тогда сообщение откладывается, пока уровень фрагмента принимающей вершины u не поднимется до нужного значения.

В конечном итоге, каждая вершина находит минимальное исходящее ребро, если такое имеется. Теперь вершины посылают сообщения *Report* (см. рис. 1.а)), чтобы найти



а) б)

Рис. 1: Схема работы алгоритма GHS. На рис. а) стрелки обозначают отправку сообщений *Report*. На рис. б) стрелка обозначает отправку сообщения *Change core* в сторону с минимальным исходящим ребром фрагмента. Ребра, которые представлены сплошными линиями, находятся в состоянии *Branch*.

минимальное исходящее ребро всего фрагмента. Если ни одна из вершин графа не имеет исходящих ребер в состоянии *Basic*, то алгоритм завершается, а ребра в состоянии *Branch* являются минимальным остовным деревом.

Отправка *Report* происходит по следующим правилам. Каждая листовая вершина фрагмента отправляет сообщение *Report(w)* по единственному инцидентному ребру в состоянии *Branch* (w — вес минимального исходящего ребра из вершины или бесконечность, если исходящих ребер нет). Каждая внутренняя вершина находит свое собственное минимальное исходящее ребро и ждет получения всех сообщений от всех поддеревьев. Затем вершина выбирает минимальный вес из всех значений весов. Если минимум достигается на значении, которое пришло из поддеревьев, то в переменную вершины *best_ edge* записывается номер исходящей ветви (поддерева), иначе записывается номер минимального исходящего ребра. Это делается для того, чтобы в дальнейшем можно было легко восстановить путь, двигаясь туда, куда указывает *best_ edge*. Далее происходит отправка *Report* вверх по дереву фрагмента с аргументом, равным уже найденному минимальному значению среди всех значений весов. Когда вершина отправляет сообщение *Report*, она также переходит в состояние *Found*. В конечном итоге, две вершины, инцидентные ядру, отправляют сообщения *Report* вдоль ядра и определяют вес минимального исходящего ребра, и с какой стороны от ядра оно находится.

Чтобы попытаться присоединить фрагмент к другому по найденному минимальному исходящему ребру фрагмента, можно воспользоваться переменной *best_ edge* в каждой вершине, чтобы проследить путь от ядра до минимального исходящего ребра. Для этого от одной из ядровых вершин, которая ближе к минимальному исходящему ребру, посылается сообщение *Change core* (см. рис. 1.б)). Вершина, получившая это сообщение, посылает его дальше в соответствии со своим значением *best_ edge*, и так далее. Когда сообщение достигает вершины с минимальным исходящим ребром, то корнем дерева, которое образует фрагмент, становится эта вершина. Эта вершина посылает сообщение *Connect(L)* по минимальному исходящему ребру, где L — это уровень фрагмента. Если два фрагмента уровня L имеют одно и то же минимальное исходящее ребро, то каждый из них отправляет сообщение *Connect(L)* вдоль этого ребра, и это ребро становится ядром нового фрагмента уровня $L + 1$, которое сразу начнет рассылать по всему фрагменту сообщение *Initiate*, с

новым номером уровня и идентификатором.

Когда фрагмент с уровнем L и идентификатором F отправляет сообщение *Connect* во фрагмент с уровнем $L' > L$ и идентификатором F', больший фрагмент отправит сообщение *Initiate* с L' и F' в меньший фрагмент.

Сложность алгоритма GHS (количество коммуникационных сообщений) — $O(N \log N)$. В данном описании алгоритма разобраны не все возникающие случаи, а лишь основные.

3. Параллельный алгоритм поиска MST на основе алгоритма GHS

Алгоритм GHS в статье 1983 года [11] представляет собой лишь описание и анализ необходимых высокоуровневых действий, которые должны выполняться в каждой вершине. Не было найдено ни одной параллельной реализации этого алгоритма, которая показывала бы хорошее масштабирование и описывала детали реализации.

Для разработки параллельного алгоритма поиска MST на основе алгоритма GHS необходимо обоснованно выбрать и разработать набор приемов, разрешить ряд проблем. Программная реализация разработанного алгоритма производилась на языке C++ с использованием библиотеки MPI [19]. При реализации на суперкомпьютере количество вершин в графе значительно больше, чем MPI-процессов, поэтому в памяти каждого процесса обычно хранится большое количество вершин и вся относящаяся к ним информация. Все вершины графа блоками распределяются последовательно по процессам. Локальная часть графа в каждом процессе хранится в формате CRS (англ. Compressed Row Storage).

3.1. Предварительная обработка исходного графа

Прежде чем искать в графе минимальное остовное дерево, над графом проводится предварительная обработка: из графа удаляются петли и кратные ребра. Удаление кратных ребер используется для выполнения условия алгоритма GHS, которое заключается в том, что все ребра должны быть уникальны. Время, затрачиваемое на предварительную обработку, незначительно и не учитывается в общем времени работы алгоритма.

3.2. Базовая версия

В начале работы над алгоритмом была разработана базовая версия. В каждом MPI-процессе поддерживается очередь, в которую вершины помещают сообщения. Для ускорения работы реализована агрегация сообщений, для каждого возможного процесса-получателя в каждом процессе заводится отдельный буфер. Схема реализации базовой версии представлена на рис. 2.

В сообщениях кроме необходимой для работы алгоритма информации содержится также служебная информация: номер вершины-отправителя и номер вершины-получателя, а также тип сообщения.

Важно отметить, что алгоритм GHS требует, чтобы исходный граф был связным. В разработанном автором алгоритме это необязательно, так как алгоритм будет работать до тех пор, пока в сети не наступит состояние «тишины», когда все очереди пусты, все сообщения обработаны, и в сети нет недоставленных сообщений. Таким образом, разработанный алгоритм позволяет находить не только MST в связном графе, но также и минимальный остовный лес в графе, с любым количеством компонент связности.

Так как алгоритм GHS требует, чтобы веса всех ребер графа были различными, вместе с обычным весом ребра хранится также специальный идентификатор $special\_id$. Для каждого ребра e графа $special\_id$ вычисляется следующим образом: пусть u и v вершины, инцидентные ребру e, тогда $special\_id$ в битовом представлении равен подряд записанным битовым представлениям $min(u, v)$, $max(u, v)$. При такой организации, даже если во

```
Input: local_G - local part of the graph
Output: local part of the MST

While (True) {
        /* read messages and push them to the queue */
        read_msgs ();
        /* queue processing, sending messages (record in the buffer) */
        If (time_to_process_queue) {
                process_queue ();
        }
        If (time_to_send) {
                /* send all accumulated messages */
                send_all_bufs ();
        }
        /* checking for completion using MPI_Allreduce */
        check_finish ();
}
```

Рис. 2: Схема реализации базовой версии параллельного алгоритма построения MST с использованием библиотеки MPI; выполняется параллельно на каждом MPI-процессе.

входном графе есть два различных ребра с одинаковым весом, алгоритм будет работать корректно.

3.3. Поиск локальных ребер

Когда MPI-процесс принял входящее сообщение, необходимо найти ребро (индекс ребра в списке локальных ребер), по которому оно пришло, то есть по двум вершинам (отправителю и получателю) в локальном списке ребер найти индекс ребра, которое образуют эти вершины. Поиск необходим для того, что может потребоваться изменение локальных данных, связанных с этим ребром.

В базовой версии для этой операции используется линейный поиск, при котором перебираются все ребра, инцидентные вершине-получателю, если вершина на другом конце ребра равна вершине-отправителю, то нужное ребро найдено.

Первым возможным вариантом оптимизации этой операции была сортировка у каждой вершины исходного графа всех инцидентных ребер в порядке увеличения номеров вершин на противоположном конце ребра. При таком подходе необходимо вначале работы алгоритма потратить немного времени на сортировку, однако во время работы алгоритма можно использовать бинарный поиск вместо линейного. Такой подход дает небольшой выигрыш по производительности.

Вторая рассмотренная оптимизация — хеширование. Вместо сортировки и бинарного поиска можно в каждом процессе создать хеш-таблицу. Пусть u — вершина-отправитель сообщения, v — вершина-получатель, каждый из номеров — машинное слово длиной 32 бита. Определим хеш-функцию $get\_hash(u, v)$ как

$$((u \ll 32) \mid v) \bmod hash\_table\_size,$$

где $\ll$ — битовый сдвиг влево, $\mid$ — побитовое или, $\bmod$ — остаток от деления, $hash\_table\_size$ — размер хеш-таблицы (в несколько раз больше, чем количество локальных ребер).

Использованный в разработанном алгоритме метод хеширования называется *линейное исследование и вставка* [20]. Таким образом, найти идентификатор локального ребра по паре смежных вершин можно за $O(1)$, но предварительно нужно создать и заполнить хеш-таблицу. Эта процедура входит в инициализацию алгоритма, занимает крайне незначительное время и не учитывается во времени работы алгоритма.

3.4. Отдельная обработка сообщений типа *Test*

Сообщения некоторых типов (*Connect*, *Test* и *Report*) не всегда возможно обработать сразу, так как для их обработки необходимо выполнение различных условий. Условие будет выполняться, когда изменятся некоторые данные, а для изменения конкретных данных необходимо дождаться определенного сообщения. Поэтому возникают ситуации, когда сообщение нужно отложить, а затем пытаться обработать снова. Когда оно будет обработано — неизвестно.

В процессе исследования работы алгоритма выявлено, что сообщения типа *Test* составляют значительную часть от всех сообщений. Оказалось, что выгодно организовать отдельную очередь для сообщений типа *Test*, и обрабатывать ее значительно реже, чем основную очередь.

3.5. Оптимизация длины сообщений

Для достижения максимально возможной производительности программной реализации на распределенной вычислительной системе необходимо минимизировать объем передаваемых по сети данных. Поэтому важно, чтобы структура, которая хранит передаваемое сообщение, занимала как можно меньший объем памяти.

Сначала проведена группировка сообщений на «короткие» (*Connect*, *Accept*, *Reject*, *Change core*) и «длинные» (*Initiate*, *Test*, *Report*). Основное отличие — в «длинных» сообщениях передается вес, а он занимает значительный объем памяти (64 бита).

В начале каждой структуры, как для «длинных», так и для «коротких» сообщений, хранится упакованное битовое поле размером 16 бит (на самом деле, необходимо только 9 бит: 3 бита на тип сообщения, 5 бит на уровень фрагмента, 1 бит на состояние вершины). Далее в каждой из структур, хранится номер вершины-отправителя и вершины-получателя (номер — машинное слово длиной 32 бита). В длинных сообщениях далее хранится расширенный вес ребра — $special\_id$ и сам вес.

Задача поиска минимального остовного дерева редко встречается в графах, у которых много ребер с одинаковым весом. Поэтому реализована следующая оптимизация: вместо того, чтобы хранить в расширенном весе $special\_id$ (склейка двух номеров вершин, суммарно 64 бита), можно хранить минимальный из номеров MPI-процессов, хранящих данное ребро, предварительно проверив, что на каждом процессе веса всех ребер различны. Действительно, если на каждом процессе веса всех ребер различны, то два различных ребра с одинаковым весом могут находиться только на разных процессах, но тогда, чтобы понять, что такие ребра различны, хватит номеров соответствующих процессов.

3.6. Параметры разработанного алгоритма

На работу программной реализации разработанного алгоритма влияют параметры, модификация которых приводит к изменению производительности:

- BUF_SIZE — размер буфера для агрегации сообщений, которые требуется отправить, в байтах (по умолчанию — 10000 байт),
- $SENDING\_FREQUENCY$ — отвечает за то, как часто сбрасывать буферы с сообщениями (по умолчанию — через каждые 5 итераций цикла *while*),
- $CHECK\_FREQUENCY$ — отвечает за то, как часто обрабатывать очередь с сообщениями типа *Test* (по умолчанию — через каждые 20 итераций цикла *while*),
- $EMPTY\_ITER\_CNT\_TO\_BREAK$ — отвечает за то, как часто проводить проверку на завершение (по умолчанию — через каждые 100000 итераций цикла *while*),

- $HASH\_TABLE\_SIZE$ — размер хеш-таблицы, измеряется в количестве элементов. По умолчанию равен $local\_actual\_m * 5 * 11/13$, где $local\_actual\_m$ — количество локальных ребер в MPI-процессе после удаления кратных ребер и петель.

4. Исследование разработанного алгоритма на кластере «Ангара-К1»

Разработка и исследование алгоритма проводились на кластере «Ангара-К1» с сетью «Ангара».

Для оценки производительности алгоритма используются RMAT графы [21]. RMAT графы хорошо моделируют реальные графы из социальных сетей и Интернета, а также являются достаточно сложными для анализа, поэтому часто применяются для оценки эффективности алгоритмов обработки графов. В данной работе рассматриваются RMAT графы со средней степенью вершины 32, количество вершин является степенью двойки. В таком графе имеется одна большая связная компонента и некоторое количество небольших связных компонент. Все графы генерируются случайным образом (вес ребра — вещественное число, в интервале (0, 1)). Масштаб графа — число, задающее количество вершин в графе. Если n — масштаб графа, то 2^n — количество вершин в этом графе. Граф с масштабом n обозначается в дальнейшем RMAT-n. Графы рассматриваемого типа используются в тесте Graph500.

Во время запусков реализации разработанного алгоритма используются заданные по умолчанию значения параметров алгоритма, перечисленные в подразделе 3.6.

4.1. Описание кластера «Ангара-К1»

Сеть «Ангара» — российская высокоскоростная коммуникационная сеть, поддерживающая топологию «многомерный тор» (возможны варианты от 1D- до 4D-тор). СБИС маршрутизатора коммуникационной сети является разработкой АО «НИЦЭВТ» и выпущена по технологии 65 нм [22]. Сеть «Ангара» отличается высокой пропускной способностью линков и низкими задержками передачи, которые соответствуют мировому уровню.

Кластер «Ангара-К1» состоит из 36 вычислительных узлов, объединенных сетью «Ангара» [22,23]. Объем оперативной памяти каждого узла — 64 ГБ. Кластер состоит из двух типов узлов. 24 узла имеют в составе по 2 процессора Intel Xeon E5-2630 (6 ядер, 2.3 ГГц). 12 оставшихся узлов — процессор Intel Xeon E5-2660 (8 ядер, 2.2 ГГц). Топология сети на кластере «Ангара-К1» — 3D-тор 3x3x4.

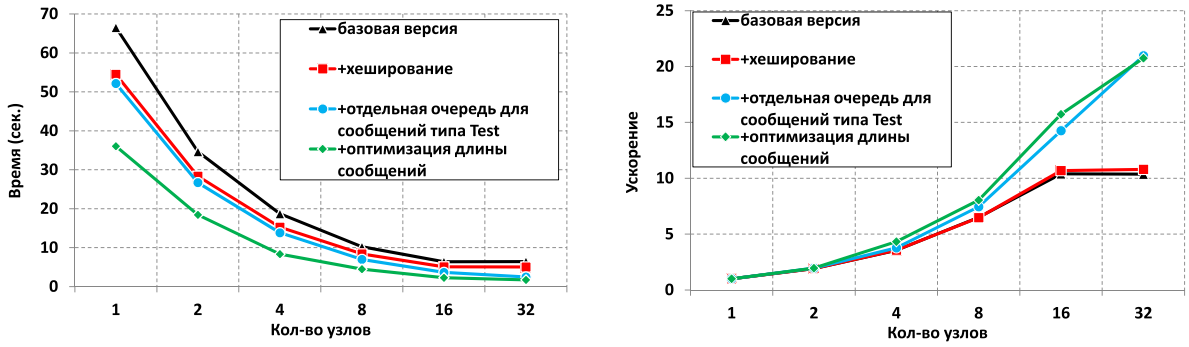
4.2. Исследование вариантов выполнения поиска локального ребра

Если для поиска локального ребра использовать бинарный поиск вместо линейного — время работы на одном вычислительном узле уменьшается на 2%, если вместо линейного поиска применить хеширование — время работы на одном узле меньше примерно на 18% (граф RMAT-23, 8 MPI-процессов на узел). Таким образом, вариант с хешированием оказался самым производительным, поэтому выбран для итоговой версии.

4.3. Сравнение производительности различных версий алгоритма

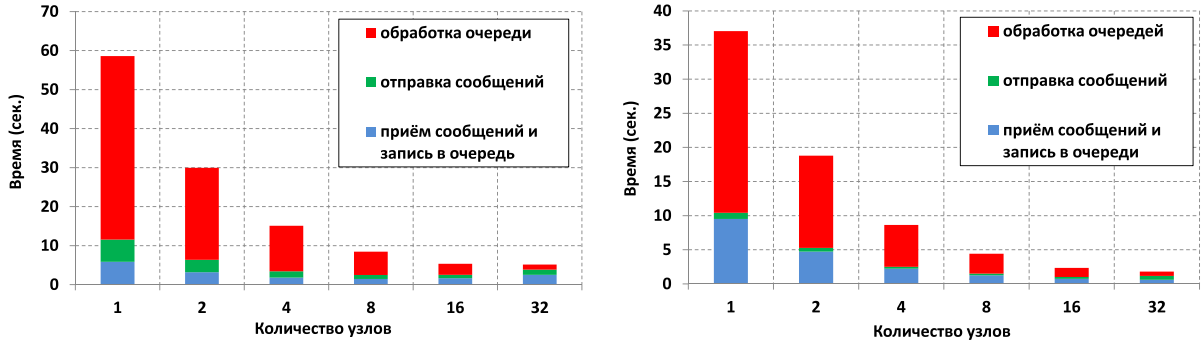
Для тестирования в данном подразделе использовался граф RMAT масштаба 23 (RMAT-23). Количество MPI-процессов на узел — 8.

На рис. 3.а) показано, как изменялось время работы программы (в секундах), по мере добавления оптимизаций, описанных в подразделах 3.3, 3.4, 3.5. На рис. 3.б) показана масштабируемость тех же самых запусков, то есть отношение времени решения задачи на одном узле ко времени решения на заданном числе узлов.



а) б)

Рис. 3: Сравнение версий: от базовой до итоговой (со всеми оптимизациями). 8 MPI-процессов на узел, граф RMAT-23. На рис. а) — время работы, на рис. б) — масштабируемость.



а) б)

Рис. 4: Результаты профилирования. На рис. а) для версии с одной оптимизацией — хешированием, на рис. б) — для итоговой версии. 8 MPI-процессов на узел, граф RMAT-23.

На рис. 4.а) представлены результаты профилирования варианта программы с одной оптимизацией поиска локального ребра, а на рис. 4.б) — результаты профилирования итоговой версии программы.

Профилирование показывает, что большая часть времени уходит на обработку очередей. Некоторые сообщения обрабатываются повторно, в том числе сообщения типа *Test*, поэтому в итоговой версии алгоритма, в которой сообщения типа *Test* обрабатываются реже, доля обработки очередей в общем времени работы алгоритма ниже, чем в варианте только с хешированием. Именно эта оптимизация позволила в 2 раза улучшить масштабируемость алгоритма, см. рис. 3.б).

Также, на производительность сильно повлияла оптимизация, связанная с уменьшением длины сообщений, которая позволила уменьшить время выполнения итоговой версии алгоритма на любом количестве узлов (от 1 до 32) приблизительно на 25%.

4.4. Исследование производительности алгоритма

В таблице 1 приведены результаты оценки производительности итоговой версии алгоритма для графа RMAT-24. Количество MPI-процессов на узел — 8.

RMAT-24 — это наибольший граф, который помещается в память одного узла кластера «Ангара-К1», размер такого графа составляет приблизительно 6.5 ГБ. Остальная оперативная память вычислительного узла необходима для организации работы реализации. В частности, большой объем памяти необходим для организации хеш-таблицы. Таблица по-

Таблица 1: Производительность разработанного алгоритма (RMAT-24).

Количество узлов	**1**	**2**	**4**	**8**	**16**	**32**
Время (сек.)	**78,70**	**39,07**	**19,63**	**9,76**	**4,87**	**2,89**
Ускорение	**1**	**2,01**	**4,01**	**8,07**	**16,18**	**27,20**

казывает, что масштабируемость близка к линейной.

Средний размер коммуникационных сообщений в зависимости от периода работы итоговой версии алгоритма показан на рис. 5. Под размером сообщения здесь понимается агрегированное (накопленное) сообщение, которое отправлено в сеть. Значение параметра агрегации BUF_SIZE составляет 20000 байт. Рисунок показывает, что с увеличением количества вычислительных узлов размер сообщений уменьшается. Для 32 узлов сообщения являются короткими, их размер не превосходит 2 КБ. Также видно, что размер сообщений зависит от периода работы алгоритма.

Данный рисунок позволяет высказать утверждение, что разработанный алгоритм предъявляет высокие требования к коммуникационной сети, так как эффективно передавать короткие сообщения возможно в высокоскоростной и высокореактивной сети с низкой задержкой и высоким темпом передачи коротких сообщений.

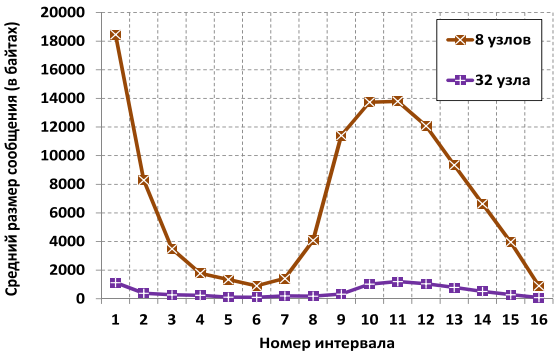


Рис. 5: Средний размер (по всем MPI-процессам) коммуникационных сообщений в байтах в зависимости от номера интервала (общее время работы алгоритма разбито на равные интервалы). 8 MPI-процессов на узел, граф RMAT-23.

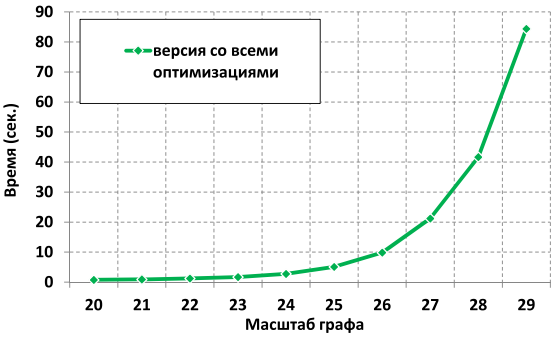


Рис. 6: Время работы итоговой версии алгоритма на графах разного размера. 32 узла. 8 MPI-процессов на узел.

Время работы на RMAT графах разного размера на 32 вычислительных узлах представлено на рис. 6. RMAT-29 — максимальный граф, который поместился в памяти 32 узлов кластера «Ангара-К1», он занимает суммарно 205 ГБ. Необходимо заметить, что разработанная реализация алгоритма решения задачи MST является масштабируемой по памяти, то есть при увеличении количества вычислительных узлов можно увеличивать размер обрабатываемого графа.

5. Заключение

В работе представлен разработанный параллельный алгоритм поиска минимального остовного дерева (леса) в графе для вычислительных систем с распределенной памятью, а также программная реализация данного алгоритма. Реализация выполнена с использованием библиотеки передачи сообщений MPI.

Впервые получены экспериментальные результаты, демонстрирующие высокий показатель масштабируемости при выполнении поиска минимального остовного дерева на вычислительных системах с распределенной памятью. На RMAT графе с 2^{24} вершин на 32 узлах кластера «Ангара-К1» достигнуто ускорение решения задачи 27.2 по отношению ко времени выполнения на одном узле.

Решение задачи для графа RMAT с 2^{29} вершин и 2^{34} ребер (для хранения такого графа необходимо приблизительно 205 ГБ памяти) получено за 84 секунды на 32 узлах кластера «Ангара-К1».

В дальнейшем планируется разработать гибридную реализацию алгоритма с одновременным использованием технологий MPI и OpenMP, а также построить аналитическую модель с целью анализа производительности алгоритма.

Работа выполнена в АО «НИЦЭВТ» под руководством к.т.н. А.С. Семенова. Автор благодарит научного руководителя на факультете ВМК МГУ им. М.В. Ломоносова к.ф.-м.н., доцента Н.Н. Попову.

Литература

1. Cormen T., Leiserson C., Rivest R., Stein C. Introduction to Algorithms. Cambridge: MIT Press and New York: McGraw-Hill, 2001. Second Edition. pp. 561–579.

2. Рубанов Л. И., Селиверстов А. В., Зверков О. А. [и др.]. Ультраконсервативные элементы у простейших из надтипа Alveolata // Современные информационные технологии и ИТ-образование. 2015. Т. 2. С. 581–585.

3. Eisner J. State-of-the-Art Algorithms for Minimum Spanning Trees. A Tutorial Discussion. University of Pennsylvania. 1997.

4. Prim R. C. Shortest connection networks and some generalizations // Bell System Technical Journal. 1957. vol. 36. pp. 1389–1401.

5. Kruskal J. B. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem // AMS. 1956. vol. 7. pp. 48–50.

6. Boruvka O. O jistem problemu minimalnim (About a Certain Minimal Problem). Prace mor. prirodoved. spol. v Brne, III, 1926. vol. 3. pp. 37–58.

7. Колганов А. С. Параллельная реализация алгоритма поиска минимальных остовных деревьев с использованием центрального и графического процессоров // Параллельные вычислительные технологии (ПаВТ'2016): труды международной научной конференции (28 марта – 1 апреля 2016 г., г. Архангельск). Челябинск: Издательский центр ЮУрГУ, 2016. С. 530–543.

8. Mariano A., Lee D., Gerstlauer A., Chiou D. Hardware and Software Implementations of Prim's Algorithm for Efficient Minimum Spanning Tree Computation // IFIP AICT. 2013. vol. 403. pp. 151–158.

9. Wang W., Huang Y., Guo S. Design and Implementation of GPU-Based Prim's Algorithm // International Journal of Modern Education and Computer Science. 2011. vol. 3. no. 4. pp. 55–62.

10. Katsigiannis A., Anastopoulos N., Nikas K. An approach to parallelize Kruskal's algorithm using Helper Threads // IEEE 26th International Parallel and Distributed Processing Symposium Workshops and PhD Forum. 2012. pp. 1601–1610.

11. Gallager R. G., Humblet P. A., Spira P. M. A distributed algorithm for minimum-weight spanning trees // ACM Transactions on Programming Languages and Systems. 1983. vol. 5. pp. 66–77.

12. Awerbuch B. Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election, and Related Problems // 19th ACM Symposium on Theory of Computing (STOC). New York. 1987. pp. 230–240.

13. Loncar V., Skrbic S. Parallel implementation of minimum spanning tree algorithms using MPI // Computational Intelligence and Informatics (CINTI), IEEE 13th International Symposium. 2012. pp. 35–38.

14. Loncar V., Skrbic S., Balaz A. Parallelization of Minimum Spanning Tree Algorithms Using Distributed Memory Architectures. Scientific Computing Laboratory, Institute of Physics Belgrade. 2014.

15. Sireta A. Comparison of parallel and distributed implementation of the MST algorithm. 2016. URL: http://delaat.net/rp/2015-2016/p41/report.pdf (accessed: 28.07.2016).

16. Ramaswamy S. I., Patki R. Distributed Minimum Spanning Trees. 2015. URL: http://stanford.edu/ rezab/classes/cme323/S15/projects/distributed_minimum_spanning_trees_report.pdf (accessed: 28.07.2016).

17. Мазеев А. В., Семенов А. С., Фролов А. С. Сравнение технологий параллельного программирования MPI и Charm++ на примере задачи построения минимального остовного дерева в графе // Comp. nanotechnol. 2015. С. 18–25.

18. McCune R. R., Weninger T., Madey G. Thinking Like a Vertex: a Survey of Vertex-Centric Frameworks for Distributed Graph Processing // ACM Computing Surveys. vol. 48. no. 2. article 25. 2015.

19. Message Passing Interface Homepage. URL: http://www.mpi-forum.org (accessed: 28.07.2016).

20. Knuth D. The Art of Computer Programming. Boston: Addison-Wesley, 1998. Second Edition. vol. 3. pp. 513–558.

21. Chakrabarti D., Zhan Y., Faloutsos C. R-MAT: A Recursive Model for Graph Mining // Proceedings of the Fourth SIAM International Conference on Data Mining. 2004. URL: http://repository.cmu.edu/cgi/viewcontent.cgi?article=1541&context=compsci (accessed: 28.07.2016).

22. Симонов А. С., Макагон Д. В., Жабин И. А. [и др.]. Первое поколение высокоскоростной коммуникационной сети «Ангара» // Наукоемкие технологии. 2014. Т. 15. С. 21–28.

23. Агарков А. А., Исмагилов Т. Ф., Макагон Д. В. [и др.]. Предварительные результаты оценочного тестирования отечественной высокоскоростной коммуникационной сети Ангара // Параллельные вычислительные технологии (ПаВТ'2016): труды международной научной конференции (28 марта – 1 апреля 2016 г., г. Архангельск). Челябинск: Издательский центр ЮУрГУ, 2016. С. 42–53.

A parallel algorithm for minimum spanning tree problem on the supercomputer with «Angara» interconnect

A.V. Mazeev

JSC «NICEVT»

The paper presents a parallel algorithm for the minimum spanning tree problem in a large graph for computer systems with distributed memory. The proposed algorithm is based on the GHS algorithm. The algorithm implementation uses C++ programming language and MPI communication librarie. The paper presents performance evaluation of the algorithm on the 36-node cluster «Angara-K1» with «Angara» interconnect.

Keywords: graphs, MST, supercomputers, «Angara» interconnect, MPI

References

1. Cormen T., Leiserson C., Rivest R., Stein C. Introduction to Algorithms. Cambridge: MIT Press and New York: McGraw-Hill, 2001. Second Edition. pp. 561–579.

2. Rubanov L. I., Seliverstov A. V., Zverkov O. A. [et al.]. Ul'trakonservativnye elementy u prosteyshikh iz nadtipa Alveolata [Ultraconservative elements in the simplest of subtype Alveolata] // Sovremennye informatsionnye tekhnologii i IT-obrazovanie [Modern information technology and IT education]. 2015. vol. 2. pp. 581–585.

3. Eisner J. State-of-the-Art Algorithms for Minimum Spanning Trees. A Tutorial Discussion. University of Pennsylvania. 1997.

4. Prim R. C. Shortest connection networks and some generalizations // Bell System Technical Journal. 1957. vol. 36. pp. 1389–1401.

5. Kruskal J. B. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem // AMS. 1956. vol. 7. pp. 48–50.

6. Boruvka O. O jistem problemu minimalnim (About a Certain Minimal Problem). Prace mor. prirodoved. spol. v Brne, III, 1926. vol. 3. pp. 37–58.

7. Kolganov A. S. Parallel'naya realizatsiya algoritma poiska minimal'nykh ostovnykh derev'ev s ispol'zovaniem tsentral'nogo i graficheskogo protsessorov [Parallel implementation of minimum spanning tree algorithm on CPU and GPU] // Parallel'nye vychislitel'nye tekhnologii (PaVT'2016): trudy mezhdunarodnoy nauchnoy konferentsii (28 marta – 1 aprelya 2016 g., g. Arkhangel'sk) [Parallel Computational Technologies (PCT'2016): Proceedings of the International Scientific Conference (28 March – 1 April 2016, Arkhangelsk)]. Chelyabinsk: SUSU Publishing Center, 2016. pp. 530–543.

8. Mariano A., Lee D., Gerstlauer A., Chiou D. Hardware and Software Implementations of Prim's Algorithm for Efficient Minimum Spanning Tree Computation // IFIP AICT. 2013. vol. 403. pp. 151–158.

9. Wang W., Huang Y., Guo S. Design and Implementation of GPU-Based Prim's Algorithm // International Journal of Modern Education and Computer Science. 2011. vol. 3. no. 4. pp. 55–62.

10. Katsigiannis A., Anastopoulos N., Nikas K. An approach to parallelize Kruskal's algorithm using Helper Threads // IEEE 26th International Parallel and Distributed Processing Symposium Workshops and PhD Forum. 2012. pp. 1601–1610.

11. Gallager R. G., Humblet P. A., Spira P. M. A distributed algorithm for minimum-weight spanning trees // ACM Transactions on Programming Languages and Systems. 1983. vol. 5. pp. 66–77.

12. Awerbuch B. Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election, and Related Problems // 19th ACM Symposium on Theory of Computing (STOC). New York. 1987. pp. 230–240.

13. Loncar V., Skrbic S. Parallel implementation of minimum spanning tree algorithms using MPI // Computational Intelligence and Informatics (CINTI), IEEE 13th International Symposium. 2012. pp. 35–38.

14. Loncar V., Skrbic S., Balaz A. Parallelization of Minimum Spanning Tree Algorithms Using Distributed Memory Architectures. Scientific Computing Laboratory, Institute of Physics Belgrade. 2014.

15. Sireta A. Comparison of parallel and distributed implementation of the MST algorithm. 2016. URL: http://delaat.net/rp/2015-2016/p41/report.pdf (accessed: 28.07.2016).

16. Ramaswamy S. I., Patki R. Distributed Minimum Spanning Trees. 2015. URL: http://stanford.edu/ rezab/classes/cme323/S15/projects/distributed_minimum_spanning_trees_report.pdf (accessed: 28.07.2016).

17. Mazeev A. V., Semenov A. S., Frolov A. S. Sravnenie tekhnologiy parallel'nogo programmirovaniya MPI i Charm++ na primere zadachi postroeniya minimal'nogo ostovnogo dereva v grafe [A comparison of MPI and Charm++ parallel programming technologies on the minimum spanning tree problem] // Comp. nanotechnol. 2015. pp. 18–25.

18. McCune R. R., Weninger T., Madey G. Thinking Like a Vertex: a Survey of Vertex-Centric Frameworks for Distributed Graph Processing // ACM Computing Surveys. vol. 48. no. 2. article 25. 2015.

19. Message Passing Interface Homepage. URL: http://www.mpi-forum.org (accessed: 28.07.2016).

20. Knuth D. The Art of Computer Programming. Boston: Addison-Wesley, 1998. Second Edition. vol. 3. pp. 513–558.

21. Chakrabarti D., Zhan Y., Faloutsos C. R-MAT: A Recursive Model for Graph Mining // Proceedings of the Fourth SIAM International Conference on Data Mining. URL: http://repository.cmu.edu/cgi/viewcontent.cgi?article=1541&context=compsci (accessed: 28.07.2016). 2004.

22. Simonov A. S., Makagon D. V., Zhabin I. A. [et al.]. Pervoe pokolenie vysokoskorostnoy kommunikatsionnoy seti «Angara» [The first generation of high-speed «Angara» interconnect] // Naukoemkie tekhnologii [Science Intensive Technologies]. 2014. vol. 15. pp. 21–28.

23. Agarkov A. A, Ismagilov T. F., Makagon D. V. [et al.]. Predvaritel'nye rezul'taty otsenochnogo testirovaniya otechestvennoy vysokoskorostnoy kommunikatsionnoy seti Angara [Preliminary results of the evaluation test of the domestic high-speed Angara

interconnect] // Parallel'nye vychislitel'nye tekhnologii (PaVT'2016): trudy mezhdunarodnoy nauchnoy konferentsii (28 marta – 1 aprelya 2016 g., g. Arkhangel'sk) [Parallel Computational Technologies (PCT'2016): Proceedings of the International Scientific Conference (28 March – 1 April 2016, Arkhangelsk)]. Chelyabinsk: SUSU Publishing Center, 2016. pp. 42–53.