УДК 004.67

Некоторые эксперименты по построению и анализу графа Де Брёйна

Марчук А.Г. (Институт систем информатики СО РАН)

Трошков С.Н. (Институт систем информатики СО РАН)

В статье описывается опыт решения задачи нахождения цепочек в графе Де Брёйна с применением параллельных вычислений и распределенным хранением данных.

Ключевые слова: граф, распределенные системы, dotnet, Spark, Scala

1. Введение

Одной из важных задач в современной биоинформатике является сборка генома по многочисленным прочтениям фрагментов ДНК. Задача, о которой пойдет речь в статье – начальный этап задачи о сборке генома. Для простоты, можно представить геном как длинную строку символов, где каждый символ соответствует азотистому основанию, из которого состоит ДНК. Аппаратура, которую используют биологи, позволяет считать множество маленьких частей этой строки, и наша цель – восстановить полную строку по ним.

Существует множество программ для сборки генома, но эффективного распределенного решения для этой задачи нет. Между тем в нём есть потребность, так как объем данных, с которыми сталкиваются биологи, может превосходить физически возможный объем данных для одного компьютера.

Авторы выражают благодарность Юрию Вяткину за помощь в постановке задачи и Денису Мигинскому за участие в обсуждении способов решения и полученных результатов.

2. Задача

Мы будем решать часть задачи сборки генома, а именно нахождение контигов [3]. Контиг представляет собой набор перекрывающихся сегментов ДНК, которые в совокупности представляют собой консенсусную область ДНК. Опуская биологические подробности, сформулируем задачу о нахождении контигов, используя математические объекты и термины.

На вход подаются строки разной длины из символов алфавита A, C, G, T.Далее выбирается некоторое число K, меньшее, чем длина самой короткой из строк. Каждая из данных строк разбивается на подстроки длины K. Эти подстроки - вершины нашего графа, он называется граф Де Брёйна. Между двумя вершинами проводится грань, если в исходной строке эти подстроки пересекаются по K-1 символам. Нужно найти все цепочки вершин без разветвлений и свернуть их в одну вершину. Получившиеся вершины называются контигами. Таким образом, по заданному набору данных секвенирования, требуется построить граф Де Брёйна и найти некоторое число наиболее длинных контигов.

3. Решение Apache Spark

Для первой группы экспериментов, в качестве платформы для разработки был выбран Apache Spark [6]. Apache Spark позволяет создавать приложения для кластерных систем, не задумываясь о многопоточности и распределении процессорных ресурсов, при условии что вы используете специальные структуры данных Spark для хранения и обработки данных.

3.1. Структуры данных Apache Spark

RDD(Resilient Distributive Dataset) Впервые представлен в версии Apache Spark 1.0. Распределенная коллекция данных, расположенных по нескольким узлам кластера, набор объектов Java или Scala, представляющих данные. RDD [5] работает со структурированными и с неструктурированными данными.

DataFrame Впервые представлен в версии Apache Spark 1.3. Распределенная структура данных, во многом похожая на таблицу в реляционной БД. Поддерживает язык запросов SQL [1].

Для решения задачи была выбрана именно DataFrame, как более современная структура с богатым набором функций и оптимизаций. В частности, при тестировании выяснилось, что DataFrame занимает примерно в два раза меньше объема оперативной памяти, чем RDD с такими же данными. К тому же, любой RDD можно перевести в DataFrame, а любой DataFrame можно перевести в RDD, хоть эта операция и может быть затратной на больших данных. Единственной проблемой при этом было то, что встроенная библиотека для работы с графами GraphX работает именно на RDD. Но существует аналог такой библиотеки для DataFrame – библиотека GraphFrames, которая в будущем, возможно, станет

частью Apache Spark.

3.2. Операции в Apache Spark

Операции в Apache Spark делятся на трансформации и действия.

Трансформация – это массовое преобразование, применяемое ко всей структуре данных. Например, фильтрация результатов по какому-либо критерию, агрегация, слияние двух структур данных, операция map. При вызове трансформации Spark запоминает ваш вызов, но вычислений не производит.

Реальные вычисления же начнутся только с вызовом действия на структуре данных. При этом Spark попытается оптимизировать порядок выполнения трансформаций, без ущерба выходным данным. К действиям относятся подсчет полей, вывод, операция reduce.

Внутри трансформаций вызывать другие трансформации нельзя. Важно, что операция cache(), которая помещает структуру данных в кэш оперативной памяти, считается трансформацией, а не действием. Это значит, что данные будут закэшированы не во время вызова операции, а лишь после вызова следующего действия.

3.3. Работа с Apache Spark

Apache Spark можно запускать на системах управления кластерами Apache Mesos, Kubernetes, YARN, либо в режиме Standalone без систем управления кластерами. Режим Standalone является самым простым для настройки, поэтому он был выбран для проведения экспериментов. Для тестирования использовались 4 виртуальные машины, для каждой машины было отведено 4 ядра процессора Inter Xeon CPU E5-2680 2.7 GHz и 20 ГБ оперативной памяти.

Для решения задачи был составлен итерационный алгоритм. На каждой итерации проверялось является ли очередная вершина началом цепочки. Если она является таковой, то происходит обход цепочки, удаление лишних вершин и переименование исходной вершины. В результате, после обхода всех вершин, получаем граф без цепочек. Схема работы алгоритма представлена на Рис. 1.

Итерировать ряды DataFrame по одному можно с помощью функции map или foreach. Эти функции являются трансформациями, а потому итерировать ряды возможно, но использовать для преобразований информацию из этого же или других DataFrame весьма проблематично. Поэтому в первой версии написанной программы итерирование произ-

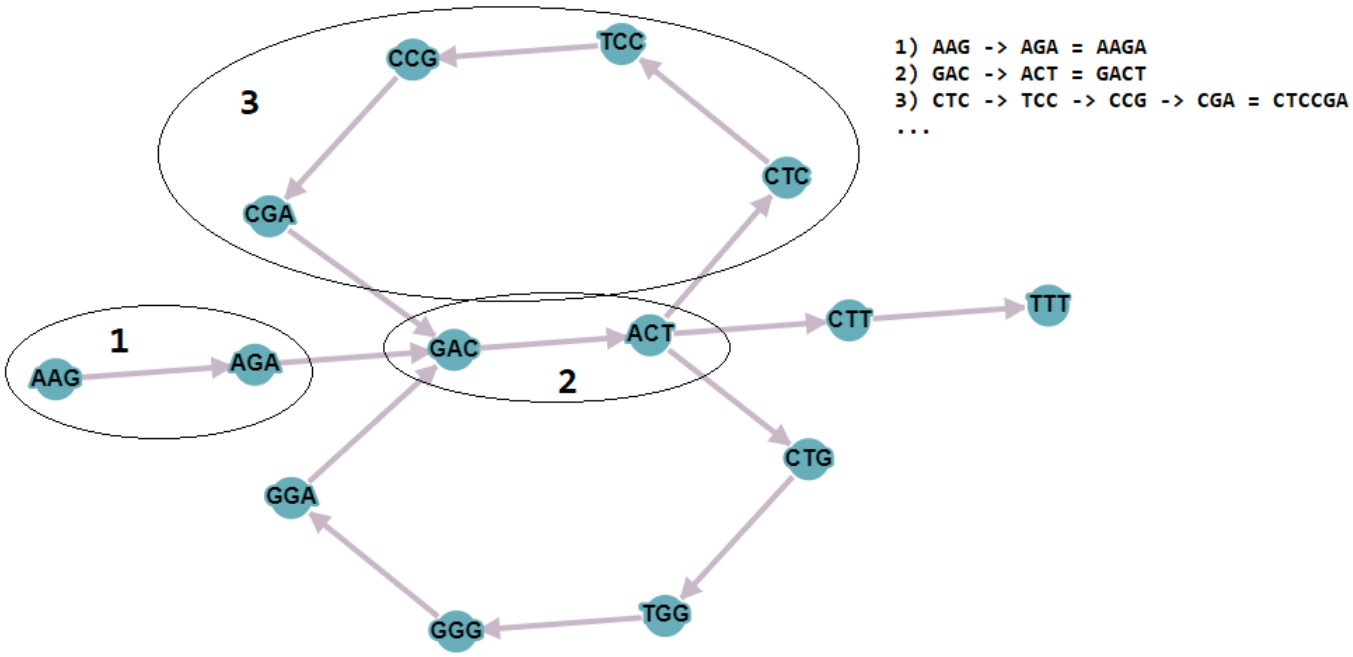

Рис. 1. Первая версия алгоритма

водилось внешним циклом языка Scala, в котором преобразовывался очередной ряд из DataFrame, в зависимости от ссылок в нем. По нагрузке процессоров было выявлено, что такая программа работает на одном компьютере, и лишь обращается к другим за данными. Это было логично, ведь задачи в Spark будут выполняться параллельно, только если это трансформации и действия над большим объемом данных. Требовалось изменить алгоритм, чтобы он работал сразу с несколькими записями, а не с каждой по очереди.

Далее описывается принцип работы новой версии алгоритма. Для начала находятся все ребра, являющиеся частью цепочки в графе. То есть, ребра, в которых у начальной вершины исходящая степень равна 1, а у конечной вершины входящая степень равна 1. После, среди найденных ребер нужно найти ребра, которые являются началом цепочки. Это будут такие ребра, исходящих вершин которых нет среди множества входящих вершин ребер, найденных нами на предыдущем шаге алгоритма. Следующим шагом нужно «склеить» вершины начальных ребер, то есть совместить их исходящую и входящую вершину в одну. Проделав эти шаги, мы укоротили все цепочки в графе на одно ребро несколькими массовыми операциями. Эти действия нужно повторять рекурсивно, пока не иссякнет множество ребер, являющихся цепочками в графе. На выходе множество вершин и будет являться искомыми контигами. Схема работы алгоритма представлена на Рис. 2.

4. Решение C#

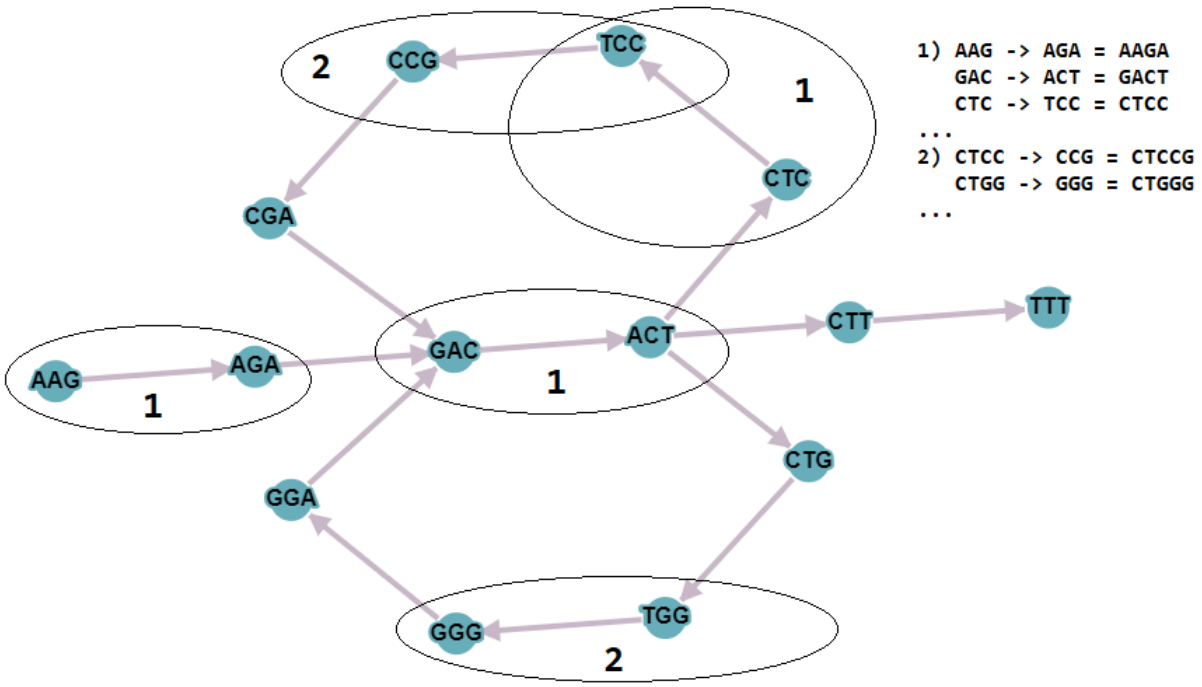

Рис. 2. Вторая версия алгоритма

Время работы программы, написанной в среде SPARK, показалось огромным для такой относительно простой задачи. Для второй группы экспериментов было принято решение написать программу на языке C#, которая решала бы эту же задачу как можно проще, не используя к сложные структуры данных, такие как Dataframe.

Для проверки предельных возможностей работы с графом Де Брёйна, были созданы две программы DeBruijnOrtho и DeBruijnDirect. Программы создавались с использованием библиотеки PolarDB [4] и с некоторым упрощением графа. Упрощение заключалось в том, что в графе не строились дуги между узлами если узел-источник дуги имеет больше одного наследника или узел-приемник дуги имеет больше одного предшественника. Полученный граф представляет собой множество линейных цепочек узлов, связанных отношением соседства. Нас интересовали стадии создания графа и вычисления максимальной (самой длинной) цепочки. Если задачу обработки графа Де Брёйна ограничить только вычислением контига максимальной длины или даже всех контигов, то указанное упрощение графа эквивалентно исходному.

Программы создавались «вручную» достаточно низкоуровневыми структурами и средствами обработки (файлы, массивы, структуры, стандартная библиотека, библиотека PolarDB). Программа DeBruijnOrtho делалась для работы в сетевой среде с использованием средств бинарного обмена информацией TCP. Для оценки пределов производи-

тельности для однозадачной (однопроцессорной) конфигурации, была сделана программа DeBruijnDirect. Существенная часть ее в какой-то мере повторяет решения DeBruijnOrtho, но полностью исключена сетевая часть и произведены специфические оптимизации.

4.1. Сетевое устройство программ

Рассмотрим некоторые особенности программ и некоторые примененные способы оптимизации. Отметим, что программа ориентирована на обработку традиционных семейств данных (ридов), возникающих в современных процессах секвенирования геномов живых организмов. Причем оптимизация велась как в сторону уменьшения используемых ресурсов, в основном памяти, так и в сторону ускорения вычислений. Предполагалось, что обработку данных уровня человеческого генома (оценочно, это 3.5 млрд. узлов графа) можно будет выполнять на одном компьютере со средним объемом ОЗУ или на относительно небольшом высокопроизводительном кластере. Большой (иногда огромный) объем начальных и промежуточных данных составляет главную проблему.

Программа организована как множество задач (процессов) обработки, расположенных на разных компьютерах. Одна из запускаемых задач называется «мастером», остальные называются исполнителями. Технически, запускается один и тот же код (программа), но на соответствующих компьютерах и с соответствующими параметрами. По параметрам, программа определяет запущена ли она как мастер или как исполнитель.

Сначала запускается мастер, ему никого ждать не надо, он сам ждет, когда с ним свяжутся. Исполнителю дается IP-адрес мастера и он устанавливает связь с ним. Связь между мастером и исполнителем работает отдельными командами, посылаемыми мастером. То есть, мастер посылает 1 байт – это команда , потом он посылает фактические значения аргументов команды и ждет результатов. Принимает результаты и на этом цикл команды заканчивается и исполнитель ждет следующую команду. Аргументов и результатов может не быть, но если они запланированы, то они обязаны появиться и в нужном формате.

Все общение производится в бинарном виде средствами BinaryWriter и BinaryReader. Посылаются любые примитивные типы (байт, целые разных размеров, строки и т.д.). Последовательность посылается посылкой двойного целого с числом элементов и потом подряд идут элементы последовательности. Запись - посылаются подряд идущими элементами записи. В общем - как в Поляре. В дальнейшем предполагается внедрить типовую систему Поляра, это позволит усилить контроль за коммуникациями.

Обработка данных выполняется на мастере. На мастере выполняются преобразования, требующие поэлементной потоковой обработки и на мастере организуются вычисления, требующие привлечения исполнителей. Исходными данными является набор ридов, задаваемый в текстовом виде во входном файле и расположенный на мастере.

4.2. Получение узлов графа

Первый этап - преобразование ридов в бинарную форму. Это потоковая обработка, когда последовательность строк ридов преобразуется в последовательность слов с заданным окном сканирования. Окно - это диапазон задаваемой длины n выборки последовательности символов. Для строки рида S первое слово будет S[0], ... S[n-1], второе - S[1], ... S[n] и так далее. Слова кодируются в двойные целые (UInt64) каждый символ - 2 разряда, от младших разрядов к старшим. Соответствие символа 2-битному коду следующее: A-0, C-1, G-2, T-3. Есть два варианта, ограничивающие размер окна в 32 и 64 символа. В дальнейшем, предполагается перейти к формату набора байтов. Таким образом, результатом первого этапа обработки является последовательность последовательностей кодированных слов. Полученный массив также располагается на мастере. В принципе, получаемый массив - довольно большой. Для набора из 500 тыс. ридов, занимающего 49 мб получается файл размером 319 мб. Для задачи в 1000 раз большей, это уже будет представлять проблему.

Следующий этап - перевод бинарных ридов в кодированные. Кодированные риды - это те же наборы слов, но закодированные целочисленным кодом. Кодирование - взаимнооднозначное, т.е. по слову однозначно определяется код, по коду - слово. Первое соответствие после второго этапа не сохраняется. Второе - сохраняется. Главное, что кодирование обеспечивает то, что 2 разных слова имеют разные коды и 2 одинаковых - одинаковые. Эффективное кодирование требует довольно много оперативной памяти, напр. для 500 тыс. ридов может потребоваться около 2 Гб. ОЗУ. Применены 2 приема для снижения нагрузки на память. Во-первых, кодирование выполняется в несколько проходов, во вторых - с привлечением исполнителей.

Основная операция, применяемая на этой стадии:

$IEnumerable < int > GetSetNodes(IEnumerable < UInt64 > bwords)$;

В ней поток бинарных слов преобразуется в поток их кодов с одновременным пополнением хеш-таблицы. На каждом из проходов преобразуется в коды лишь часть слов, соответственно объем хеш-таблицы уменьшен. Причем по завершению прохода, мы мо-

жем уничтожить таблицу имен потому что в оставшейся части таких слов не остается. Многопроходный подход "конвертирует"дополнительное время на обработку в уменьшение расхода оперативной памяти. Использование исполнителей основывается на том, что в каждом исполнителе накапливается набор слов, обладающих некоторыми свойствами типа word%n-1 == номер секции (исполнителя). Соответственно, в указанном методе GetSetNodes набор слов разбивается по секциям и выполняется обращение к этим секциям, а полученные результы сливаются. В результате данного этапа формируется последовательность (для кластерного варианта - множество последовательностей) узлов, в которой номер элемента последовательности является кодом слова. Соответственно, на этом этапе, слова превращаются в узлы.

Другая часть графа - ребра, реализуется (в упрощенном представлении) через ссылки (коды узлов) "вперед"и "назад в структуре узла CNode. Это делается через модификацию списка узлов при сканировании файла кодированных ридов. Следующий этап – нахождение начал непрерывных цепочек. Основная идея в том, что цепочка может начинаться только с узла у которого нет предыдущего или предыдущих несколько или предыдущий один, но у него несколько следующих. Кроме того, нет смысла рассматривать те узлы, у которых не единственный следующий. В нормальных данных число непрерывных цепочек (контигов) существенно меньше числа узлов, поэтому такой список оперативную память не нагружает.

Далее, идет этап отслеживания цепочек с выделением самой длинной. Собственно идет продвижение от начального узла по указателям на следующий до пропажи указателя. Если при этом, будет превышено количество пройденных узлов в данной цепочке, то лучшая цепочка заменяется на текущую. Главная проблема процесса прохождения цепочек – медленность чтения следующего указателя из последовательности узлов графа, поскольку очередной узел может располагаться в произвольном месте последовательности, а значит – файла последовательности. Ускорение этого процесса осуществляется запуском одновременного отслеживания большого числа цепочек с последующим группированием запросов на получение следующего узла. Что это дает? Если граф разбит на секции, а секции располагают узлы в более «близких» местах, напр. на одном вычислителе, то сгруппированные запросы снова разбиваются теперь уже по секциям и выполняются более эффективно и, возможно, в параллель.

5. Сравнение двух подходов

В таблицах ниже представлено общее время работы программ на кластере в зависимости от размера входных данных и количества виртуальных машин в кластере, работающих над задачей. В работе мы использовали как реальные входные данные, так и специально сгенерированные для тестирования. Конкретно, 500 тысяч ридов и 14 миллионов ридов взяты из реальных ДНК вируса свиного гриппа и бактерии E. Coli соотвественно. 5 и 10 миллионов ридов – данные, которые были получены искусственно, с помощью генератора.

Программа, написанная с помощью Apache Spark, не работала со сгенерированными данными и с меньшим количеством машин. В этих случаях сильно вырастала нагрузка на сеть, и временные файлы для передачи данных переполняли всю физическую память машин.

Решение C#	500 тыс. ридов	5 млн. ридов	10 млн. ридов	14 млн. ридов
1 Машина	44s	5m 10s	10m 42s	15m 4s
2 Машины	1m 32s	8m 54s	17m 3s	23m 46s
4 Машины	1m 44s	11m 40s	22m 32s	29m 29s
8 Машин	2m 26s	19m 57s	33m 52s	38m 30s
16 Машин	5m 16s	76m 31s	111m 16s	88m 45s

Решение Apache Spark	500 тыс. ридов	14 млн. ридов
8 Машин	3h 24m	5h 18m
16 Машин	2h 35m	3h 1m

Все тесты проводились на кластере, собранном из виртуальных машин. Каждая из виртуальных машин находилась физически на одном и том же жестком диске, и сеть между ними была виртуальная, поэтому в наших решениях мы опустили оценку скорости передачи данных по сети.

Несмотря на то, что решение Apache Spark работает в разы медленнее, именно на нём можно заметить рост производительности с увеличением количества машин, то есть выигрыш от распределенных вычислений. Время работы программы на C# увеличивается с добавлением машин, но всё же значительно превосходит Apache Spark по времени, и при работе на одной машине, оно даже сравнимо с известным решением SPAdes [2]. В нашем тесте на 500 тысяч ридов программа SPAdes нашла контиги за 15 минут, 13 из которых занимал обязательный этап исправления ошибок в ридах. Так как в нашей программе не учитываются разные биологические эвристики и нет этапа исправления ошибок, можно

предположить, что наше решение будет не хуже.

Список литературы

1. Amburst M., Xin R. Spark SQL: Relational Data Processing in Spark // SIGMOD '15: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. May, 2015. P. 1383–1394.
2. Bankevich A., Nurk S. SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing // Journal of computational biology : a journal of computational molecular cell biology, №19. 2012. P. 455-477.
3. Gregory S. Contig Assembly // Encyclopedia of Life Sciences. -2005.
4. Marchuk A.G. PolarDB: an infrastructure for specialized NoSQL databases and DBMS // Automatic Control and Computer Sciences. - 2016. - V. 50. - № 7. - P. 493-496.
5. Matei Zaharia, Mosharaf Chowdhury. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing // University of California, Berkeley. Technical Report No. UCB/EECS-2011-82. -2011.
6. Srinivas Jonnalagadda V., Srikanth P. A Review Study of Apache Spark in Big Data Processing // International Journal of Computer Science Trends and Technology -2016. - V. 3. - № 4. - P. 93-98.