PARALLEL IMPLEMENTATION OF SOME ALGORITHMS OF COUNTING THE NUMBER OF NONISOMORPHIC FINITE SEMIGROUPS

E.I. Kuzichkina, postgraduate student
D.I. Vlasov, student
Togliatti state university, Togliatti (Russia)

Keywords: finite semigroup; exhaustive search; heuristics; distributed computing; parallel algorithms.

Annotation: Considered heuristic search algorithms for calculation of the number of finite semigroups and options for parallelization.

УДК 519.178

НЕКОТОРЫЕ ПОДЗАДАЧИ ЗАДАЧИ ВЕРШИННОЙ МИНИМИЗАЦИИ НЕДЕТЕРМИНИРОВАННЫХ КОНЕЧНЫХ АВТОМАТОВ

М.В. Кукеев, аспирант
Тольяттинский государственный университет, Тольятти (Россия)

Ключевые слова: базисный автомат; покрывающий автомат; параллельная модель вычисления CUDA.

Аннотация: В статье рассматриваются подзадачи нахождения циклов базисного автомата и проверка их наличия в покрывающем автомате, а также алгоритмизация этих подзадач для параллельной модели вычисления.

ВВЕДЕНИЕ

Решение задачи минимизации детерминированного конечного автомата Рабина-Скотта было придумано уже достаточно давно. Тогда казалось, что это является конечной точкой в теории регулярных языков. Однако, со временем, стала очевидна необходимость использования и других формализмов в теории регулярных языков. Недетерминированный конечный автомат оказался очень удобным средством во многих прикладных задачах теории регулярных языков. Например, недетерминированные конечные автоматы активно используются для построения компиляторов, также удобно использовать недетерминированность в построении доказательств теорем в теории регулярных языков. Системы поиска слов, подстрок, предложений в разных приложениях используют недетерминированные конечные автоматы, как удобное средство построения таких алгоритмов поиска.

В современных условиях возникают задачи построения минимального по какому-либо критерию недетерминированного конечного автомата. Вершинная минимизация позволяет минимизировать состояния конечного автомата. Это полезно, например, для компактного представления этого автомата в памяти компьютера.

Так как для задачи вершинной минимизации недетерминированных конечных автоматов пока не существует эффективных алгоритмов, имеет смысл пытаться улучшить существующие алгоритмы или написать вспомогательные, которые позволят в какой-нибудь степени расширить возможности практического применения этих алгоритмов. Для расширения практического применения можно попытаться уменьшить время исполнения алгоритмов за счет применения различных технологий. Например, можно распараллелить алгоритм для его исполнения в параллельных моделях вычислений.

В данной работе используется модель параллельного вычисления, основанная на архитектуре CUDA (Compute Unified Device Architecture – унифицированная архитектура вычислений) [1], которая является перспективной во многих научных направлениях. И, хотя появилась на рынке эта архитектура сравнительно недавно (примерно в 2006 году), уже сейчас она активно внедряется в научные «круги».

Использование архитектуры CUDA в задачах с объёмными вычислениями часто даёт приличный прирост производительности, а если уж задача изначально располагает к параллелизму, то равных архитектуре CUDA по параметру цена/производительность нет.

ОБЩИЕ ПОНЯТИЯ

Под минимизацией конечного автомата понимают нахождение эквивалентного конечного автомата,

минимального по какому-либо критерию. Эквивалентность автоматов означает то, что эти автоматы задают один и тот же язык (множество слов, принимаемых (распознаваемых) конечным автоматом). При минимизации эквивалентность является фундаментальным условием, так как нам необходимо, чтобы минимальный автомат задавал тот же язык, что и исходный. Иначе, просто теряется смысл такой минимизации.

В данной работе рассматриваются некоторые подзадачи вершинной минимизации для недетерминированных конечных автоматов. Такой критерий минимальности подразумевает минимально возможное число состояний автомата, естественно, с сохранением эквивалентности. В теории вершинной (и не только) минимизации недетерминированных конечных автоматов используются такие понятия, как базисный автомат, функция разметки, блок, покрывающий автомат и многие другие. Подробное описание этих понятий, доказательства свойств и других немаловажных вещей выходит за рамки этой работы [2].

ВЫДЕЛЕНИЕ ПОДЗАДАЧ

При вершинной минимизации недетерминированного конечного автомата, использующей покрывающие автоматы, возникает подзадача определения, задаёт ли полученный покрывающий автомат тот же язык, что и исходный автомат.

Существует очевидный алгоритм такой проверки. Мы строим по полученному покрывающему автомату канонический и сравниваем с полученным ранее каноническим автоматом с точностью до переобозначения состояний. Как известно, построение канонического автомата не эффективно с точки зрения сложности алгоритма, да и не хотелось бы строить заново канонический автомат, поэтому в реальных ситуациях желательно использовать другие методы. Полезным бывает доказать, что данный покрывающий автомат не задаёт исходный регулярный язык.

Для определения того, что найденный автомат не задает исходный язык, будем найденные циклы, не включающие в себя подциклы, базисного автомата «искать» (в программе – проверять) в полученном автомате. Если цикл отсутствует – автомат не подходит, и следует выбрать другое покрывающее множество блоков. Это утверждение верно, так как если в покрывающем автомате отсутствует хотя бы один цикл, то найдётся слово данного языка, прочитывая которое, базисный автомат «проходит» цикл (причём, может быть и не один раз, в зависимости от выбранного слова), отсутствующий в покрывающем автомате. Разбив это слово на три части: преффикс, без учёта той части, которая распознаётся рассматриваемым циклом, часть, распознаваемая циклом, и оставшийся суффикс, и прочитав преффикс, базисный автомат окажется в одном или нескольких из своих состояний. Значит, покрывающий автомат, по определению, должен оказаться в одном или нескольких из своих состояний, включающих состояния, в которых оказался базисный автомат. Аналогично с суффиксной частью. Ввиду однозначности базисного автомата существует только единственная последовательность тактов базисного автомата, распознающая данное слово, значит «верным» окажется только одно единственное состояние базисного автомата, в которое он попадёт после прочтения преффикса, и с которого будет начато чтение суффикса. Следовательно, опираясь на построение покрывающего автомата, мы можем утверждать, что он также содержит этот цикл.

При больших размерностях базисного автомата, нахождение всех циклов и их проверка в покрывающем автомате выполняется достаточно долго, поэтому было бы полезно ускорить этот процесс. Это мы и попытаемся сделать, используя параллельную архитектуру CUDA.

АНАЛИЗ ПОДЗАДАЧ ДЛЯ ИХ ДАЛЬНЕЙШЕЙ АЛГОРИТМИЗАЦИИ

Используя предыдущий пункт, мы выделили две обособленные, но связанные подзадачи. Первая – нахождение циклов базисного автомата, вторая – проверка наличия этих циклов в покрывающем автомате.

Рассмотрим первую подзадачу.

Так как конечные автоматы могут быть очень просто представлены в виде ориентированных графов, имеет смысл рассматривать конечный автомат с точки зрения теории графов. Тогда, следуя предложенному, переформулируем задачу в теории графов.

Дан ориентированный граф с помеченными дугами. Требуется найти множество всех его циклов, не включающих подциклы. Такая задача уже давно решена. Существует несколько алгоритмов её решения. Однако, до сих пор не существует эффективного алгоритма её решения, поэтому имеет смысл попытаться придумать эвристический подход, позволяющий в какой-то степени облегчить поиск всех циклов.

Вообще говоря, первый алгоритм, рассматриваемый мной, в теории графов придуман для неориентированных графов. Однако, делая небольшие поправки в этот алгоритм, мы получаем требуемый результат. Опишем сначала исходный вариант алгоритма [3, с. 95].

Шаг 1: строим остовное дерево на основе нашего графа.

Шаг 2: перебираем все дуги, не вошедшие в остовное дерево.

Шаг 3: добавляя одну дугу к дереву, получаем в точности один цикл, не содержащий подциклов.

Шаг 4: выписываем получившиеся циклы.

Сложность этого алгоритма без учёта шага 4 – $O(m * n)$ [4, с. 91], где m – количество дуг, а n – количество вершин. Под сложностью здесь понимается асимптотичесткая (т. е. в пределе) верхняя граница.

В нашем случае шаг 4 подразумевает нахождение всех циклов графа, проходящих через рассматриваемую дугу. Сложность именно этого шага является камнем преткновения для эффективного решения этой (нахождение всех циклов графа) задачи.

При необходимости, можно ослабить требование нахождения всех циклов. В случае с рассматриваемым алгоритмом, мы, например, можем эффективным способом найти циклы, получающиеся при добавлении к остовному дереву отсутствующих в нём дуг.

Допустим, мы нашли покрывающий автомат и начинаем проверку наличия в нём циклов базисного автомата. Если все циклы, найденные эффективным способом, присутствуют в покрывающем автомате, можно последовательно искать остальные ещё не найденные циклы и сразу же проверять их наличие в покрывающем автомате. Конечно, такой подход не позволяет существенно улучшить эффективность, но в некоторых случаях позволит сузить пространство поиска.

Так как понятие остовного дерева для ориентированного графа не определено, следует пояснить значение шага 1. Мы хотим найти такой подграф, содержащий все вершины исходного графа, что добавление любой не вошедшей в него дуги образует цикл (в терминах ориентированного графа – контур). Это позволит нам не рассматривать всё множество вершин. Забегая вперёд, можно сказать, что мы будем искать выходящее дерево (другое название – корневое дерево) [5, с. 235] в нашем графе, причём максимальное. То есть, невозможно добавить дугу к дереву, не нарушив его свойств. Таким образом мы минимизируем количество дуг, не вошедших в корневое дерево. Корневое дерево – это ориентированный граф с источником, не имеющий полуконтуров [5, с. 233]. Источником в графе называется вершина, из которой можно достичь все другие вершины графа.

А теперь, собственно, покажем, почему этот алгоритм в исходном виде не подходит для нашего графа. В случае неориентированного графа нам всё равно, откуда начинать строить остовное дерево, однако в ориентированном графе далеко не из каждой вершины можно обойти весь граф. А нам это необходимо для нахождения корневого дерева. Значит, в нашем случае имеет значение, откуда начинать построение корневого дерева. Так как базисный автомат не содержит бесполезных и недостижимых состояний, то, очевидно, начав построение со стартовых состояний, мы гарантированно пройдём по всем возможным вершинам и дугам. Следовательно, вершины, соответствующие всем стартовым состояниям базисного автомата, образуют источник. Есть ещё несколько нюансов, появившихся в результате ориентированности графа: если в неориентированном графе мы пытались присоединить к текущему дереву уже присоединённую вершину, то сразу же натыкались на образование цикла. Это очевидно, ведь обе рассматриваемые вершины входят в дерево, а значит, существует путь между этими вершинами. Стоит только добавить дугу, соединяющую эти вершины, мы сразу получаем цикл. В ориентированном графе в вершину может входить сколько угодно дуг, циклы от этого не появятся. Важно, выходят ли из этой вершины дуги; если да, то при тех же условиях, что и в неориентированном графе, возможно появление цикла. Но для этого должен существовать путь между рассматриваемыми вершинами. В ориентированном графе это далеко не очевидно: если две вершины входят в дерево, это не значит, что между ними существует путь, это значит, что точно существует путь из начально рассмотренной вершины в эти вершины. Следовательно, возникает вопрос, как тогда определять появление цикла. Выше мы уже оговорились, что будем строить максимальное корневое дерево. Для его построения, фактически, используется простой алгоритм для нахождения остовного дерева для неориентированного графа [4, с. 646]. После работы такого алгоритма мы получим лишние дуги, то есть такие, добавление которых к корневому дереву не приводит к появлению контура. Чтобы не делать повторных вычислений, немного модифицируем задачу. В определении корневого дерева заменим понятие полуконтур на контур. Соответственно, изменим и алгоритм для решения обновлённой задачи.

Для этого нам понадобится введение для каждой вершины множества вершин, из которых есть путь в эту вершину. Тогда, рассматривая очередную дугу из одного состояния в другое, смотрим наличие вершины, в которую ведёт добавляемая дуга, в множестве вершин вершины, из которой выходит рассматриваемая дуга. Если такая вершина есть в множестве, то значит уже существует путь из добавляемой вершины в текущую, и добавлять дугу нельзя. В противном случае добавляем дугу и обновляем множество вершин, из которых мы можем попасть в рассматриваемую, для всех вершин, в которые существует путь из добавленной вершины. Таким образом, мы гарантированно построим корневое дерево с вышеперечисленными поправками для ориентированного графа, построенного на основе базисного автомата. Данный алгоритм затрачивает дополнительное время при выполнении по сравнению с предыдущим, но существенным образом это не влияет на эффективность. Гораздо важнее то, что мы избавились от лишних дуг. Ведь дальше нам потребуется для каждой такой дуги искать контуры, проходящие через неё. С учётом неэффективности всех алгоритмом поиска всех контуров графа, лишние дуги гораздо сильнее замедлят работу программы, нежели дополнительные вычисления в изменённом алгоритме поиска корневого дерева.

На третьем шаге описанного алгоритма мы добавляем дугу, не попавшую в остовное дерево, и получаем в точности один цикл. Это утверждение верно только для неориентированного графа. В нашем случае может существовать более одного пути между произвольными вершинами корневого дерева. При добавлении дуги ищем все возможные циклы. Чтобы найти все возможные циклы, недостаточно рассмотрения построенного корневого дерева, так как оно может не содержать все возможные пути между двумя произвольными вершинами ориентированного графа. Это возможно, так как в состояние может входить несколько дуг, а значит и количество путей в это состояние из какой-нибудь вершины может быть больше одного.

Найдя все возможные дуги, добавление которых приводит к образованию цикла, будем искать в исходном графе все пути из одной вершины в другую, соответствующие рассматриваемой дуге. Для поиска пути можно использовать, например, алгоритм поиска в ширину или в глубину, сложность которых составляет O(m + n), где m – количество дуг, а n – количество вершин графа, в котором осуществляется поиск. Важно то, что сложность алгоритмов поиска в ширину и в глубину дана для нахождения одного пути в графе.

Рассмотрим вторую подзадачу.

При её решении никаких трудностей не возникает. Для каждого найденного контура, проверяем его наличие в покрывающем автомате. Сложность проверки каждого контура – O(n), где n – количество состояний базисного автомата. Оценка верна, так как при проверки контура мы делаем m + 1сравнений, где m – количество вершин в контуре. Очевидно, что m <= n, а значит оценка точна. Если общее количество найденных контуров равно k, то конечная сложность – O(n * k).

РЕАЛИЗАЦИЯ ПОДЗАДАЧ В «ПАРАЛЛЕЛЬНОМ» КОДЕ

Для проверки циклов в найденном автомате самым очевидным является использование одного блока выполнения графического акселератора для проверки одного цикла. Блок должен состоять из количества потоков, равных количеству состояний базового автомата. Текущее ограничение на количество потоков в одном блоке составляет 512, что достаточно неплохо. Каждый поток имеет доступ к найденному автомату через константную память (возможно размещение в другой памяти – в зависимости от размерности автомата). Константная память кэшируема, что позволяет увеличить быстродействие, но зато сильно ограничена в размере (64 КБайта на текущий момент). Если размер автомата не позволяет использовать константную память, следует разместить его в глобальной памяти. В случае «не нахождения» цикла поток достаточно просто устанавливает общую переменную в ложь, и процесс останавливается (если необходимо).

Поиск циклов базисного автомата – более сложная задача с точки зрения параллелизации. Можно использовать простую схему: для каждого состояния базисного автомата запускается поток выполнения графического акселератора, который ищет циклы, содержащие это состояние. Поток может использовать разные алгоритмы поиска циклов (например, в ширину или в глубину). Параллелизация при таком подходе простая и легко реализуется. Недостатки очевидны: каждый цикл будет найден m раз, где m – количество состояний в цикле. Отсюда увеличение потребления памяти и снижение скорости (с большой вероятностью незначительное).

Можно использовать алгоритм, описанный в предыдущих пунктах. Начнём с построения корневого дерева. Распараллелить алгоритмы построения остовного дерева сложно, да и эффекта от этого будет мало (алгоритмы

построения остовного дерева (тем более не минимального) достаточно эффективны). Алгоритм начинает работу в стартовых состояниях базисного автомата (их (стартовых состояний) может быть не больше, чем $2^n - 1$, где n – число строк таблицы отношения #) и добавляет дуги к текущему корневому дереву. Если «новая» добавляемая дуга не образует контур, добавляем её и обновляем множество вершин для каждой из рассмотренных вершин, в которые мы можем попасть по добавленной дуге. Продолжаем, пока можем. Так как у нас дуги, нам при добавлении важны не входящие в вершины дуги, а исходящие (ведь от того, что в состояние ведут несколько дуг, циклы не появятся). Для нахождения корневого дерева можно использовать параллельность за счёт большего, чем 1 количества стартовых состояний базового автомата. Каждый поток выполнения графического акселератора начинает работу в «своём» стартовом состоянии базисного автомата. Так как базисный автомат не содержит бесполезных и недостижимых состояний, алгоритм завершит работу корректно. Однако есть особенности: все потоки должны быть в одном блоке (блок – набор потоков, связанных между собой разделяемой памятью и синхронизацией). Размер блока ограничен (512 потоков в настоящее время). Это ограничение не является критическим: мы можем запустить алгоритм построения остовного дерева несколько раз, если количество стартовых состояний базисного автомата превосходит 512. Например, если стартовых состояний 758, то мы сначала «прогоняем» первые 512, а на втором этапе – оставшиеся 246. Сложность этого алгоритма без учёта параллелизации – $O(n + m)$, где n – количество состояний базисного автомата, а m – количество дуг базисного автомата. С учётом параллелизации производительность должна быть повыше.

Не вошедшие в корневое дерево дуги образуют пары состояний, через которые проходят искомые циклы. Тогда, используя для каждой такой пары блок графического акселератора размером в количество потоков равное количеству состояний базисного автомата, ищем путь из одного состояния этой пары в другое состояние этой пары (имеет значение начальное состояние пути и конечное; определяем по рассматриваемой дуге), и, добавляя соответствующую дугу, получаем контур. Важным здесь является то, что пути мы ищем не в корневом дереве, а в исходном базисном автомате, так как в данном случае корневое дерево не гарантирует нахождение всех циклов таким способом. Для поиска пути можно воспользоваться, например, алгоритмом поиска в ширину или в глубину. Сложность соответствует сложности алгоритма поиска пути.

ЗАКЛЮЧЕНИЕ

В качестве продолжения данной работы предлагается провести статистический анализ работы описанных алгоритмов на последовательной и параллельной моделях вычислений.

Работа автора частично поддержана программой Министерства образования и науки в рамках Госзадания Тольяттинского государственного университета на 2012 (шифр 6.3072.2011), а также региональным грантом РФФИ № 13-01-97003.

СПИСОК ЛИТЕРАТУРЫ

1. Берилло, А. NVIDIA CUDA – неграфические вычисления на графических процессорах [Электронный ресурс]. URL: http://www.ixbt.com/video3/cuda-1.shtml (дата обращения: 07.10.2009).
2. Мельников Б. Ф. Недетерминированные конечные автоматы ; монография. – Тольятти: ТГУ, 2009. – 160 с.
3. Липский В. Комбинаторика для программистов ; пер. с пол. В. А. Евстигнеева и О. А. Логиновой ; под ред. А. П. Ершова. – М.: Мир, 1988. – 200 с.
4. Кормен Т. [и др.] Алгоритмы: построение и анализ ; пер. с англ.. – 2-е изд. – М.: Вильямс, 2005. – 1296 с.
5. Харари Ф. Теория графов ; пер. с англ. В. П. Козырева ; под ред. Г. П. Гаврилова. – 2-е изд. – М.: Едиториал УРСС, 2003. – 296 с.

SOME PROBLEMS OF STATE-MINIMIZATION OF NONDETERMINISTIC FINITE AUTOMATA

M.V. Kukeev, postgraduate student
Togliatti State University, Togliatti (Russia)

Keywords: basis automaton; covering automaton; parallel compute architecture CUDA.

Annotation: In this article we take look at problems to find all cycles in basis automaton and check its(cycles) existence in covering automaton. Also we build algorithms for these problems in parallel architecture.