МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Кафедра вычислительных технологий**

**КУРСОВАЯ РАБОТА**

**ПАРАЛЛЕЛЬНАЯ РЕАЛИЗАЦИЯ АЛГОРИТМА РЕШЕНИЯ ЗАДАЧИ О НАИБОЛЬШЕЙ КЛИКЕ**

Работу выполнила \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Е.В.Степанова

(подпись, дата) (инициалы, фамилия)

Факультет компьютерных технологий и прикладной математики 3 курс

Направление 02.03.02 – «Фундаментальная информатика и информационные технологии»

Научный руководитель, доц.

канд. физ-мат. наук, доц. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Е.В.Кособуцкая

(подпись, дата) (инициалы, фамилия)

Нормоконтролер, доц.

канд. физ-мат. наук, доц. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Е.В.Кособуцкая

(подпись, дата) (инициалы, фамилия)

Краснодар 2017

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc501202745)

[1 Задача о наибольшей клике 5](#_Toc501202746)

[2 Алгоритмы решения задачи о наибольшей клике 7](#_Toc501202747)

[2.1 Алгоритм простой итерации 7](#_Toc501202748)

[2.2. Алгоритм расширенной итерации с удалением 7](#_Toc501202749)

[2.3 Алгоритм с учётом перекрытий 8](#_Toc501202750)

[2.4 Алгоритм Брона-Кербоша 8](#_Toc501202751)

[2.5 Алгоритм Хансера 10](#_Toc501202752)

[2.6 Выбор алгоритма 11](#_Toc501202753)

[3. Технологии параллельного программирования 12](#_Toc501202754)

[4. Реализация алгоритма Брона-Кербоша 14](#_Toc501202755)

[4.1 Оценка возможности распараллеливания 14](#_Toc501202756)

[4.2 Детали реализации 17](#_Toc501202757)

[4.2.1 Генерация графа для исследования 17](#_Toc501202758)

[4.2.2 Варианты распределения вычислений между потоками 18](#_Toc501202759)

[4.3 Результаты работы 20](#_Toc501202760)

[ЗАКЛЮЧЕНИЕ 24](#_Toc501202761)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 25](#_Toc501202762)

[ПРИЛОЖЕНИЕ А Последовательная реализация алгоритма Брона-Кербоша на языке С++ 27](#_Toc501202763)

[ПРИЛОЖЕНИЕ Б Текст процедуры, генерирующей случайный граф на для заданного количества вершин и рёбер 28](#_Toc501202764)

[ПРИЛОЖЕНИЕ В Текст процедуры, выполняющей статическое распределение вычислений для алгоритма Брона-Кербоша 29](#_Toc501202765)

[ПРИЛОЖЕНИЕ Г Текст процедуры, выполняющей динамическое распределение вычислений для алгоритма Брона-Кербоша 31](#_Toc501202766)

# ВВЕДЕНИЕ

Методы теории графов применимы во многих областях науки: электронике, химии, экономике, социологии, лингвистике, генетике.

Графы используются для моделирования реальных систем таких как распределительные и компьютерные сети, что приводит к графам большого размера. Для анализа подобных графов необходимы эффективные алгоритмы с использование параллельных вычислений.

Задача о наибольшей клике (Maximum Clique Problem, MCP) является одной из известных NP-трудных задач теории графов, для которой пока не найдено алгоритмов, разрешающих эту задачу за полиномиальное время. Данная задача имеет многочисленные приложения. В биоинформатике MCP используется при анализе геномных баз данных. В социальных сетях MCP применяется при кластеризации данных – при разделении различных сообществ на группы (кластеры), обладающие общими свойствами. Выделение кластеров позволяет обрабатывать каждый из них отдельным вспомогательным сервером. В химии задача MCP лежит в основе поиска «максимальной общей подструктуры» в графе, описывающем структуру химического соединения. В указанных приложениях, как правило, необходимы точные решения для MCP. При этом объем входных данных огромный (входные графы могут содержать до миллиона вершин). Таким образом, актуальным направлением исследования MCP является разработка новых подходов нахождения точных решений с учетом особенностей графов, возникающих в приложениях.

Пусть есть граф G со множеством вершин V и множеством рёбер E. Тогда кликой графа называется подграф, построенный на подмножестве V1 вершин графа, являющийся полным и максимальным в том смысле, что любой другой подграф графа G, построенный на некотором множестве вершин, содержащем V1, не является полным. В клике все вершины попарно смежны.

На первый взгляд кажется, что нахождение всех клик графа – простая задача, которую можно решить последовательным перебором и запоминанием максимальных множеств. Это справедливо для небольших графов, например, с числом вершин, не превосходящих 20. Но с увеличением числа вершин этот метод становится громоздким с вычислительной точки зрения.

Одним из наиболее эффективных известных на сегодняшний день алгоритмов, решающих задачу поиска всех клик, является алгоритм Брона-Кербоша. Сложность этого алгоритма равна 3v/3, где v – количество вершин графа [4]. Однако для графов с большим количеством вершин даже этот алгоритм имеет высокую вычислительную сложность, поэтому имеет смысл выполнить параллельную реализацию алгоритма.

Целью курсовой работы является параллельная реализация алгоритма Брона-Кербоша, позволяющего решить задачу о максимальной клике.

# 1 Задача о наибольшей клике

Подмножество вершин графа называется кликой, если любые две вершины из этого подмножества смежны.

Отметим, что понятие клики графа G является противоположным к понятию независимого множества.

Если графа и S является кликой графа G, то подграф, порождённый множеством S, является полным. То есть клика — это полный подграф графа G. В графе может существовать не единственная клика.

Максимальная клика графа может быть не наибольшей.

Задача поиска наибольшей клики связана с другими задачами теории графов, такими как задача о наименьшем покрытии (ЗНП), поиск наибольшего независимого множества вершин, поиск наибольшего паросочетания (наибольшее независимое множество рёбер), поиск наибольшего доминирующего множества вершин и рёбер (рисунок 1).

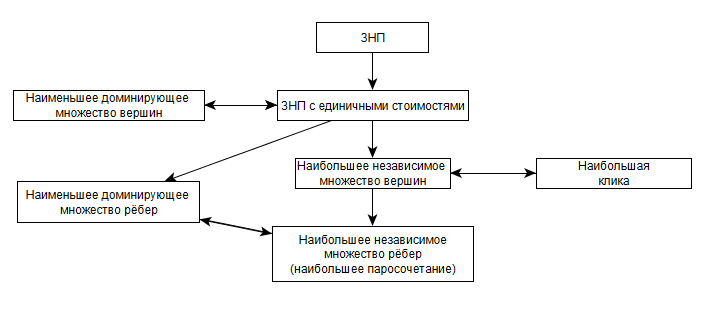


Рисунок 1 – Взаимосвязь между задачами теории графов [10]

Например, так как подмножество вершин графа G является кликой тогда и только тогда, когда оно независимо в дополнительном графе, задача о наибольшем независимом множестве вершин сводится к задаче о наибольшей клике и наоборот.

Иногда в прикладных задачах не требуется поиск точного решения задачи о наибольшей клике. От найденной клики требуется являться наибольшей лишь тогда, когда другие клики заметно меньше неё [6]. При этом решение будет приближённым.

Напротив, под точным решением понимается именно наибольшая клика, а точным алгоритмом называется такой алгоритм, который гарантированно найдёт клику с наибольшей размерностью.

В работе ставится задача разработать программу, которая на основе алгоритма Брона-Кербоша для заданного графа с количеством вершин |V| ≤ 200 и количеством рёбер |E| ≤ 2000 найдёт все клики, размер наибольшей из них, а также саму наибольшую клику (или все клики максимальной размерности, если таких клик в графе несколько).

# 2 Алгоритмы решения задачи о наибольшей клике

## 2.1 Алгоритм простой итерации

На каждом шаге алгоритма ищется вершина v с наименьшей степенью deg(v), и если deg(v) < |G|-1, где |G| - количество вершин графа на текущем шаге, то вершина v удаляется. Иначе все вершины графа имеют степень |G|-1, и полученный подграф является полным подграфом исходного графа.

В большой доле реальных задач этот алгоритм находит максимальный полный подграф, однако это не всегда так. Если существует только одна массивная клика (то есть клика, заметно превышающая по размеру остальные), то алгоритм её найдёт. Если массивных клик несколько, то алгоритм найдёт одну из них. [6]

Временная асимптотическая оценка сложности составляет O(N2), где N – количество вершин графа. [6]

Данный алгоритм не является точным и неприменим в задачах, где требуется точное решение.

## 2.2. Алгоритм расширенной итерации с удалением

На каждом шаге алгоритма производится поиск полного подграфа по алгоритму простой итерации. Затем из графа удаляются вершины, которые принадлежат найденному полному подграфу. То есть на каждом шаге поиск ведётся по тем вершинам, которые не вошли ни в один найденный полный подграф. Когда количество оставшихся вершин оказывается меньше количества вершин в наибольшем найденном полном подграфе. Этот подграф и считается искомым.

Этот алгоритм, в отличие от алгоритма простой итерации позволяет решать большее число задач без явно выраженных массивных клик, однако трудности составляют случаи, когда существуют вершины, принадлежащие одновременно нескольким кликам. Эти вершины будут приписаны ровно одному подграфу, что исказит мощность остальных клик, которым принадлежит эта вершина. [6]

Этот алгоритм также не является точным.

## 2.3 Алгоритм с учётом перекрытий

Производится поиск подграфов по алгоритму расширенной итерации с удалением. Для каждого найденного подграфа на исходном графе производится перебор вершин, которые не были включены в этот подграф. Если такая вершина соединена со всеми вершинами из найденного подграфа, она включается в этот подграф. Наибольший из найденных таким образом подграфов считается искомым.

Этот алгоритм расширяет возможности алгоритмов простой итерации, однако для алгоритма остаются неразрешимыми задачами, когда ни один из найденный алгоритмом расширенной итерации с удалением полных подграфов не лежит полностью в наибольшей искомой клике или если оставшаяся после удаления перекрытий часть клики мала по сравнению с самими перекрытиями. [6] Поэтому данный алгоритм не является точным.

## 2.4 Алгоритм Брона-Кербоша

Этот алгоритм является существенно упрощённым перебором, использующим дерево поиска. Для работы алгоритма необходимо три множества. Обозначим cand – множество вершин, которые могут быть использованы для расширения клики, not – множество вершин, которые уже были использованы для расширения клики, и compsub – множество, содержащее вершины, предположительно входящие в искомую клику.

Описание работы алгоритма.

1. Положить множества not и compsub пустыми, множество cand – множество всех вершин графа.

2. Пусть во множестве compsub k вершин. Взять вершину из множества cand, и поместить её во множество compsub.

3. Если в not есть вершина, связанная со всеми из cand, то перейти к шагу 5, иначе к шагу 4.

4. Если cand и new пусты, что напечатать клику compsub и перейти к шагу 5. Если пусто только cand, то перейти к шагу 5. Иначе перейти к шагу 2.

5. Удалить из compsub вершину . Удалить эту же вершину из cand и добавить её в not. Если cand и compsub пусты, алгоритм останавливается. Иначе перейти к шагу 3.

При программной реализации как правило используется рекурсивная функция.

Реализация процедуры на псевдокоде:

ПРОЦЕДУРА extend (candidates, not):  
ПОКА candidates НЕ пусто И not НЕ содержит вершины, СОЕДИНЕННОЙ СО ВСЕМИ вершинами из candidates,   
ВЫПОЛНЯТЬ:  
1. Выбрать вершину v из candidates и добить ее в compsub  
2. Сформировать new\_candidates и new\_not, удаляя из candidates и not вершины, не соединённые с v  
3. ЕСЛИ new\_candidates и new\_not пусты  
4. ТО compsub – клика  
5. ИНАЧЕ рекурсивно вызывать extend (new\_candidates, new\_not)  
6. Удалить v из compsub и candidates и поместить в not [5]

Для наглядности блок-схема алгоритма этой процедуры приведена на рисунке 2.

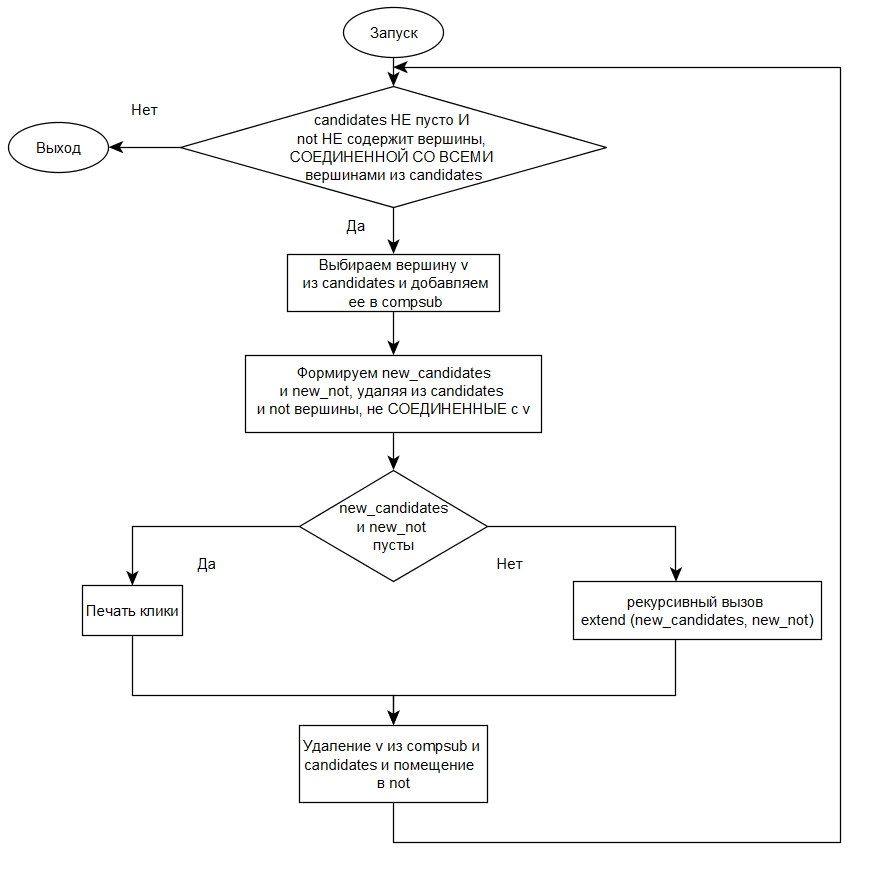


Рисунок 2 – Блок-схема алгоритма Брона-Кербоша

Алгоритм Брона-Кербоша является одним из самых эффективных известных на данный момент точных алгоритмов решения задачи о наибольшей клике.

## 2.5 Алгоритм Хансера

Алгоритм Хансера является модификацией алгоритма Брона-Кербоша. В этом алгоритме максимальность найденного множества вершин определяется только невозможностью его расширения в то время как в методе Брона-Кербоша максимальность проверяется также условием пустоты множества not (множества вершин, которые уже были использованы для расширения). Необходимым и достаточным условием в методе Хансера является то, что множество cand на некотором шаге оказывается пустым.

Отличие алгоритма Хансера от алгоритма Брона-Кербоша также состоит в изменении условия для осуществления шага возвращения. На очередном шаге может быть построено множество вершин, которое потенциально может стать наибольшей кликой . Условие возвращения заключается в проверке, является ли текущее «потенциальное множество» подмножеством к какому-нибудь из уже найденных решений. Это условие оказывается более эффективным, чем условие, проверяемое в методе Брона-Кербоша. [7]

## 2.6 Выбор алгоритма

Среди рассмотренных алгоритмов есть как приближённые, но работающие за полиномиальное время, так и точные, но требующие высоких затрат вычислительных ресурсов для больших графов. Выбор алгоритма для решения задачи зависит от самой задачи. Если скорость вычисления важнее точности, то лучше выбрать приближённый быстрый алгоритм, если же требуется точное решение, то необходимо выбрать точный алгоритм.

В работе для параллельной реализации выбран алгоритм Брона-Кербоша с целью оценки возможности сокращения времени его выполнения.

# 3 Технологии параллельного программирования

Наиболее известными технологиями параллельного программирования являются технологии OpenMP и MPI.

При использовании многопроцессорных вычислительных систем с общей памятью предполагается, что имеющиеся в составе системы процессоры обладают равной производительностью, правами и скоростью обращения к общей памяти.

Обычный подход при использовании таких систем – создание новых параллельных методов на основе последовательных программ, где компилятором или программистом выделяются участки независимых вычислений. Однако активно развивается ещё один подход, когда указания программиста добавляются при помощи директив или комментарием, обрабатываемых специальным препроцессором до начала компиляции. Этот подход и является основой технологии OpenMP.

В рамках этой технологии директивы параллелизма используются для выделения параллельных фрагментов.

Технология OpenMP позволяет эффективно реализовать возможности многопроцессорных вычислительных систем с общей памятью. Она также позволяет согласовать сложность разработки параллельной программы со сложностью решаемой задачи (распараллеливание простых программ не требует больших усилий). При этом параллельная программа с использованием этой технологии, как правило, будет работать для разных вычислительных систем с общей памятью [8].

Одной из распространённых технологий параллельного программирования является технология MPI. Основным способом взаимодействия между параллельными процессами является передача сообщений друг другу. Стандарт MPI фиксирует интерфейс, который должен соблюдаться как системой программирования, так и пользователем при создании программ.

Интерфейс MPI поддерживает создание параллельных программ, подразумевающих объединение процессов с различными исходными текстами, однако на практике чаще используется подход, где для всех параллельных процессов используется один и тот же код. Такой подход существенно упрощает написание и отладку параллельных программ. [9]

Технология MPI используется, как правило, в распределённых системах, состоящих из нескольких удалённых друг от друга вычислительных устройств. В таких системах накладные расходы существенно растут за счёт времени, затрачиваемого на пересылку данных от одного устройства к другому.

В данной работе для параллельной реализации была выбрана технология OpenMP, поскольку она встроена в Visual C.

# 4 Реализация алгоритма Брона-Кербоша

## 4.1 Оценка возможности распараллеливания

Алгоритм Брона-Кербоша реализован в виде рекурсивной процедуры (приложение А).

Рассмотрим и информационные зависимости в пределах одного вызова процедуры, а именно начального, где множества compsub и not пусты, а множество cand равно множеству всех вершин графа. Операции, происходящие при рекурсивных вызовах, рассматриваться не будут. Очевидно, что при первом вызове клика будет найдена только в том случае, если есть изолированная вершина. При поиске зависимостей эта ситуация учитываться не будет.

Реализованная процедура содержит 6 параметров:

1) cand – множество вершин, которые могут быть использованы для расширения клики.

2) not – множество вершин, которые уже были использованы для расширения клики.

3) compsub – множество, содержащее вершины, предположительно входящие в искомую клику.

4) lst – списки смежности искомого графа (способ хранения графа)

5) SetCompsub – список найденных клик.

6) Size – размер наибольшей найденной клики.

Также в процедуре есть следующие локальные переменные:

1) v – исследуемая вершина

2) new\_cand - множество вершин, которые могут быть использованы для расширения клики, при этом связанные с исследуемой вершиной v. Используется при рекурсивном вызове процедуры.

3) new\_not – множество вершин, которые уже были использованы для расширения клики, при этом связанные с исследуемой вершиной v. Используется при рекурсивном вызове процедуры.

Списки смежности исходного графа передаются по константной ссылке, то есть они не меняются и не порождают никаких информационных зависимостей.

Переменные SetCompsub и Size относятся к выходным данным. Они изменяются синхронно при обнаружении клики, поэтому при поиске информационных зависимостей их можно считать одним объектом. Следует отметить, что при изменении порядка нахождения клик поменяется лишь порядок клик в списке SetCompsub, но с точки зрения логической реализации SetCompsub – это множество и порядок элементов не имеет значения. Отметим также, что изменение этих переменных на некоторой итерации цикла while происходит тогда и только тогда, когда на этой итерации не происходит рекурсивного вызова, поэтому можно считать, что эти две переменные также не порождают новых информационных зависимостей.

Для остальных данных граф информационных зависимостей представлен на рисунке 2.

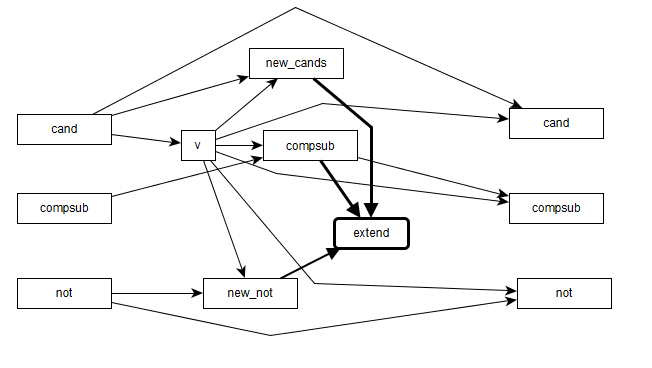


Рисунок 3 – Граф информационных зависимостей для одной итерации цикла while процедуры extend

Разделение на потоки в пределах одной итерации неэффективно, так как вычисления внутри цикла относительно простые, и накладные расходы на распараллеливание могут ликвидировать ускорение, получаемое за счёт параллельной реализации. Поэтому для большей наглядности графа зависимостей объединим вершины в группы как показано на рисунке 4.

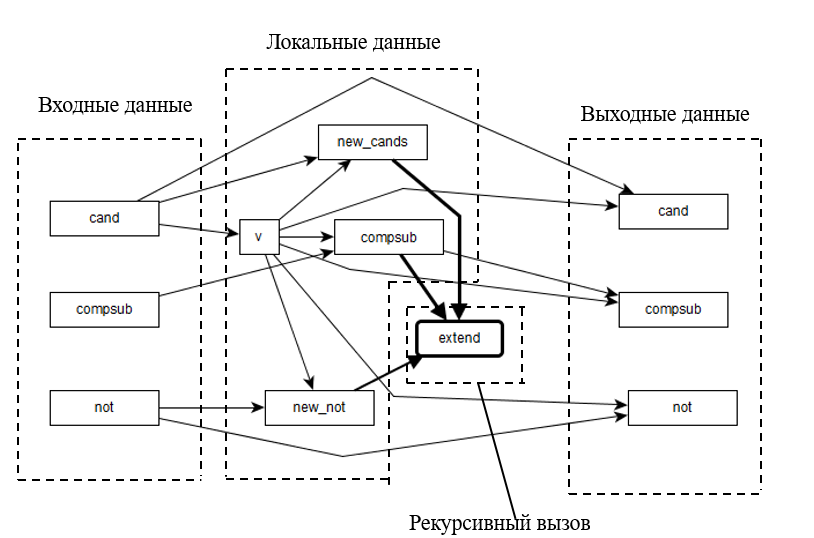


Рисунок 4 – Объединение вершин в группы

Название группы «Локальные данные» весьма условно, поскольку у неё входит список compsub, который не является локальным по отношению к итерации цикла, однако в начале итерации в него добавляется вершина v, а в конце – извлекается, поэтому этот список при переходе к следующей итерации остаётся таким же, каким был при переходе к текущей.

Также следует отметить, что «Выходные данные» для данной итерации являются «Входными данными» для следующей итерации. Поэтому, если соединить несколько таких графов для нескольких последовательных итераций, объединим группы «Входные данные» и «Локальные данные» для каждой из них и назовём эту группу «Итерация», то получим граф, представленный на рисунке 5.

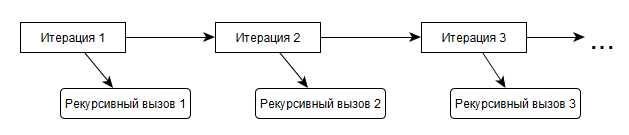


Рисунок 5 – Граф информационных зависимостей для нескольких итераций цикла

На рисунке 5 видно, что рекурсивные вызовы на разных итерациях одного цикла не имеют информационных зависимостей между собой, следовательно, они могут быть выполнены параллельно.

## 4.2 Детали реализации

### 4.2.1 Генерация графа для исследования

Граф генерируется случайным образом с помощью специально написанной для этой цели функции. На вход этой функции подаётся требуемое число вершин и рёбер. Граф представляется в виде списков смежности, то есть для каждой вершины создаётся список, куда будут вноситься смежные с ней вершины при генерации графа. Затем функция генерирует два псевдослучайных числа a и b, являющиеся предполагаемыми концами нового ребра. Затем проверяется, не является ли это ребро петлёй (то есть должно выполняться условие неравенства a и b) и не содержится ли оно уже в графе (для этого достаточно просмотреть, не содержится ли вершина b в списке смежности вершины a). Если оба условия выполнены, то в список смежности вершины a добавляется вершина b и в список смежности вершины b добавляется вершина a, а ребро считается сгенерированным. Генерация пар чисел будет происходить до тех пор, пока не будет сгенерировано нужное число рёбер. Сложность этой процедуры не включена в исследование, поскольку в практических задачах графы, как правило, получаются в результате исследований, а не генерируются в программе. Текст процедуры генерации можно посмотреть в приложении Б. Результат генерации записывается в файл, откуда его потом считает основная программа.

### 4.2.2 Варианты распределения вычислений между потоками

В данной работе распределение вычислений будет производиться на первой ступени вызова процедуры, реализующей алгоритм Брона-Кербоша. То есть поток, получив некоторую ветвь поиска при рекурсивном вызове, выполняет её один и полностью. На рисунке 6 стрелками обозначаются рекурсивные вызовы, а пунктиром выделены области, которые могут быть отданы различным потокам.

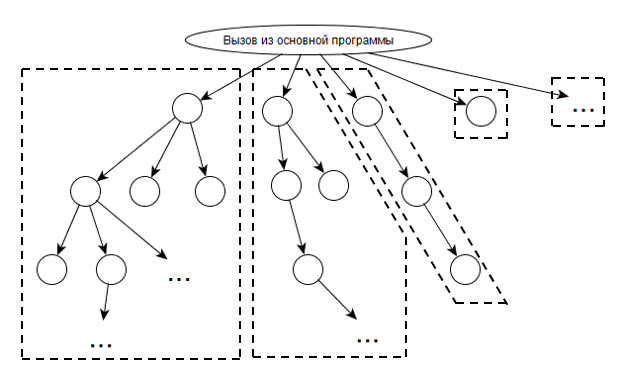


Рисунок 6 – Области дерева поиска, которые могут быть отданы различным потокам

Будет рассмотрено два метода распределения вычислений. Статический – когда заранее известно, какую ветвь какой поток будет вычислять, и динамический – когда очередную ветвь возьмёт поток, освободившийся первым.

Итерации цикла при «корневом» запуске процедуры можно пронумеровать с нуля, а рекурсивному вызову (ветви дерева поиска) на данной итерации, если он будет, присвоить номер этой итерации. Отметим, что все вычисления, производимые в цикле «корневого» запуска, кроме собственно рекурсивных вызовов, будут выполнены всеми потоками, если при этом будет найдена клика, она будет найдена всеми потоками, но в результирующий список её запишет только один поток.

При статическом распараллеливании i-ую ветвь будет вычислять поток с номером (i mod num\_threads), где num\_threads – это количество потоков. Текст процедуры, выполняющей статическое распараллеливание, представлена в приложении В.

При динамическом распараллеливании для учёта просмотренных или находящихся в обработке ветвей вводится переменная ClNum, проинициализированная нулём. Эта переменная означает номер следующей ветви, которая должна поступить в обработку. В цикле «корневого» запуска каждый поток считает итерации цикла и на каждом шаге сравнивает номер полученной итерации и переменной ClNum. Если они равны, то данная ветвь ещё не обработана, и поток увеличивает переменную ClNum на 1 и выполняет эту ветвь, иначе рекурсивный вызов не выполняется, и поток переходит к следующей итерации. Код процедуры, выполняющей динамическое распараллеливание, представлен в приложении Г.

Для каждого из этих двух вариантов распараллеливания было использовано две реализации, отличающиеся распределением памяти и расположением критических секций. Критические секции необходимы для того, чтобы при работе с общими ресурсами не возникало коллизий, например, при динамическом распараллеливании при сравнении номера итерации с переменной ClNum необходимо организовать критическую секцию для случая, когда оба потока могут посчитать ветку не проверенной и оба выполнят один и тот же рекурсивный вызов.

В первом варианте реализации всем потокам выделяется один список, куда все потоки записывают найденные клики, и при записи клик организована критическая секция. Во втором варианте каждому потоку выделяется свой список, и критическая секция необходима только в «корневом» цикле при распределении вычислений для потоков. При этом для поиска наибольшей клики необходимо найти наибольшие клики, найденные каждым из потоков и среди них выбрать ту, которая имеет максимальный размер.

## 4.3 Результаты работы

Результаты исследования представлены в таблице 1, в которой V – количество вершин графа, E – количество рёбер графа, последовательный – время выполнения последовательного алгоритма, статический – время выполнения параллельного алгоритма со статическим распределением объёма вычислений, динамический - время выполнения параллельного алгоритма с динамическим распределением объёма вычислений, общий – время выполнения при использовании общего списка клик, раздельные – время выполнения при использовании отдельных списков клик для каждого потока.

Таблица 1 – Результаты распараллеливания

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| V | E | Последо-  ватель-  ный, c | Статический, c | | Динамический, c | |
| общий | раздельные | общий | раздельные |
| 10 | 20 | 0,040917 | 0,044859 | 0,055888 | 0,057234 | 0,049422 |
| 20 | 40 | 0,129406 | 0,151544 | 0,178954 | 0,159819 | 0,179958 |
| 20 | 100 | 0,722131 | 0,492717 | 0,488234 | 0,466222 | 0,467378 |
| 30 | 40 | 0,205987 | 0,305616 | 0,297783 | 0,30949 | 0,30679 |
| 30 | 100 | 0,539501 | 0,623378 | 0,653567 | 0,739369 | 0,610133 |
| 30 | 250 | 7,13949 | 3,9047 | 3,50233 | 3,1202 | 3,36899 |
| 30 | 400 | 49,9474 | 33,4428 | 32,6395 | 30,2838 | 29,7256 |
| 50 | 70 | 0,486476 | 0,789768 | 0,812084 | 0,663757 | 0,65843 |
| 50 | 200 | 1,79129 | 1,72396 | 1,78411 | 1,60788 | 1,62531 |
| 50 | 500 | 20,0856 | 13,437 | 13,1075 | 11,7523 | 11,83 |
| 70 | 100 | 1,07888 | 1,6009 | 1,66917 | 1,28885 | 1,3582 |
| 70 | 200 | 2,02406 | 2,25425 | 2,24387 | 2,27425 | 2,31432 |
| 70 | 500 | 10,9843 | 7,88118 | 7,84987 | 7,53856 | 7,74825 |
| 70 | 1000 | 133,614 | 79,8092 | 87,1394 | 80,6573 | 82,5599 |
| 200 | 500 | 13,7541 | 16,6855 | 16,5941 | 15,0207 | 15,0717 |
| 200 | 1000 | 34,4543 | 29,6819 | 31,2322 | 33,5439 | 29,4277 |
| 200 | 2000 | 150,86 | 100,775 | 100,641 | 104,864 | 96,6475 |

Вычисления производились на компьютере с четырёхъядерным процессором с тактовой частотой 3,2 ГГц. Распределение осуществлялось между четырьмя потоками.

Для каждого графа время засекалось пять раз: один раз для последовательного алгоритма и четыре раза для параллельного (для статического распределения вычислений с общим списком и раздельными списками и для динамического с общим списком и раздельными списками). Для каждого способа распараллеливания на каждом графе было вычислено ускорение как отношение времени выполнения последовательного алгоритма на этом графе к времени выполнения параллельного алгоритма на этом же графе. В таблице 2 представлено ускорение для каждого графа для каждого способа параллельной реализации. В ней используются те же обозначения, что и в таблице 1.

Таблица 2 – Полученное ускорение для всех вариантов распараллеливания

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| V | E | Статический | | Динамический | | Максимальное ускорение |
| общий | раздельные | общий | раздельные |
| 10 | 20 | 0,912125 | 0,732125 | 0,714907 | 0,827911 | 0,912125 |
| 20 | 40 | 0,853917 | 0,723124 | 0,809703 | 0,71909 | 0,853917 |
| 20 | 100 | 1,46561 | 1,479067 | 1,548899 | 1,545068 | 1,548899 |
| 30 | 40 | 0,674006 | 0,691735 | 0,665569 | 0,671427 | 0,691735 |
| 30 | 100 | 0,865448 | 0,825472 | 0,729678 | 0,884235 | 0,884235 |
| 30 | 250 | 1,828435 | 2,038497 | 2,288151 | 2,119178 | 2,288151 |
| 30 | 400 | 1,493517 | 1,530275 | 1,649311 | 1,680282 | 1,680282 |
| 50 | 70 | 0,615973 | 0,599046 | 0,732913 | 0,738842 | 0,738842 |
| 50 | 200 | 1,039055 | 1,004024 | 1,114069 | 1,102122 | 1,114069 |
| 50 | 500 | 1,494798 | 1,532375 | 1,709078 | 1,697853 | 1,709078 |
| 70 | 100 | 0,673921 | 0,646357 | 0,837087 | 0,794345 | 0,837087 |
| 70 | 200 | 0,897886 | 0,90204 | 0,88999 | 0,874581 | 0,90204 |
| 70 | 500 | 1,393738 | 1,399297 | 1,457082 | 1,417649 | 1,457082 |
| 70 | 1000 | 1,674168 | 1,533336 | 1,656564 | 1,618389 | 1,674168 |
| 200 | 500 | 0,824315 | 0,828855 | 0,915676 | 0,912578 | 0,915676 |
| 200 | 1000 | 1,160785 | 1,103166 | 1,027141 | 1,170812 | 1,170812 |
| 200 | 2000 | 1,496998 | 1,498991 | 1,438625 | 1,56093 | 1,56093 |

Видно, что для некоторых графов, как правило, с небольшим количеством рёбер, ускорение не было достигнуто. Это связано с тем, что для небольших графов объём вычислений невелик, но накладные расходы на распараллеливание значительны. Однако для некоторых графов ускорение довольно существенное, а в одном из экспериментов наблюдалось ускорение более, чем в два раза. Соотношение количества экспериментов, в которых последовательный алгоритм отработал быстрее и количества экспериментов, в которых одна из параллельных реализаций оказалась быстрее, представлено на рисунке 7.

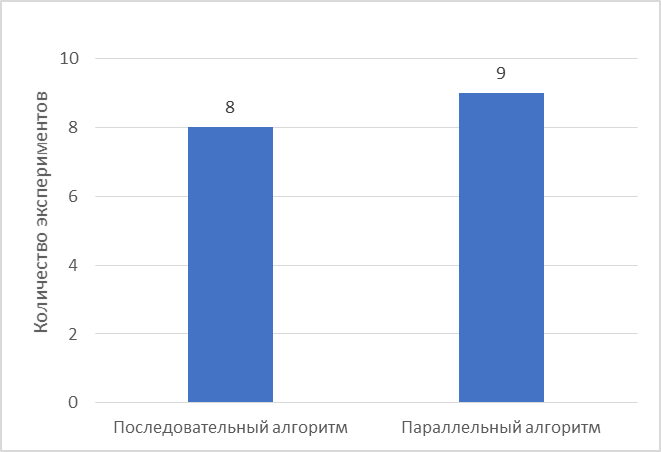


Рисунок 7 – Количество экспериментов, в которых последовательный алгоритм оказался эффективнее параллельного и наоборот.

Видно, что разница невелика и практически в половине случаев ускорения не было достигнуто. Это связано с тем, что рекурсивные алгоритмы плохо поддаются распараллеливанию.

Имеет смысл сравнить различные способы распределения вычислений между потоками. Количество экспериментов для каждого способа распараллеливания, в которых этот способ оказался быстрее остальных, представлен на рисунке 8.

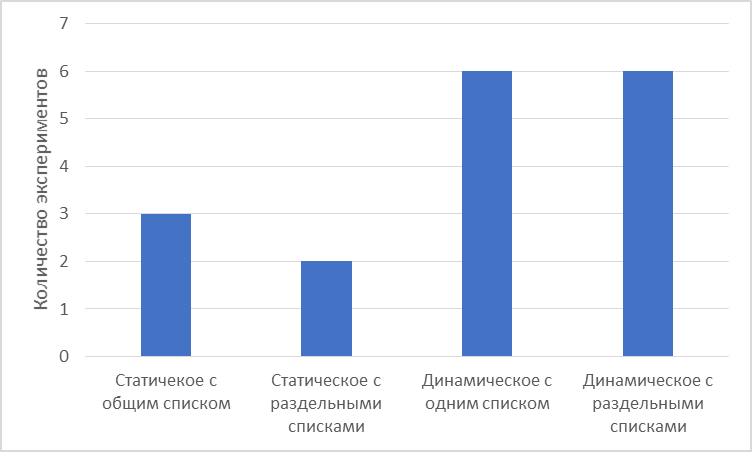


Рисунок 8 – Сравнение количества экспериментов, в которых каждый способ распараллеливания оказался быстрее остальных.

Видно, что в большинстве случаев динамическое распараллеливание оказалось эффективнее, однако ввиду небольшого числа экспериментов, однозначного вывода сделать нельзя.

# ЗАКЛЮЧЕНИЕ

В процессе работы были изучены различные алгоритмы нахождения наибольшей клики, их достоинства и недостатки, а также возможности использования технологии OpenMP для практической реализации параллельного алгоритма на языке С++ в среде Visual Studio 2015.

Было рассмотрено два варианта распределения вычислений между потоками: статический и динамический. Однозначного вывода о том, какой из этих способов лучше по результатам исследований сделать нельзя, в каких-то случаях алгоритм со статическим распараллеливанием работает быстрее, в каких-то – с динамическим.

В дальнейшем планируется усовершенствование принципов распределения вычислений между потоками, например, разделение объёмов вычислений не только на первой ступени рекурсии. Также планируется проверка алгоритма на различных типах графов, моделирующих реальные объекты.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Реализации алгоритмов/Алгоритм Брона — Кербоша [Электронный ресурс], URL: https://ru.wikibooks.org/wiki/Реализации\_алгоритмов/ Алгоритм\_Брона\_-\_Кербоша [Дата обращения 3 декабря 2017].
2. Описание алгоритма Брона Кербоша по нахождению максимальной клики [Электронный ресурс], URL: https://github.com/Elizaveta Kozlovskykh/Bron-Kerbosch-algorithm/wiki/ [Дата обращения 3 декабря 2017].
3. Н. Кристофидес Теория графов. Алгоритмический подход. ; изд-во «Мир», 1978. – 432 c.
4. Илларионов Р. Е. Двухфазный алгоритм решения задачи о клике для разреженных графов большой размерности // Молодой ученый. — 2016. — №3. — С. 4-8. — URL: https://moluch.ru/archive/107/25826/ [Дата обращения: 16.12.2017].
5. Алгоритм Брона — Кербоша (для нахождения клик) [Электронный ресурс]. URL: <http://lmatrix.ru/news/theory/algoritm-brona-kerbosha-dlya-nakhozhdeniya-klik_125.html> [Дата обращения 16 декабря 2017]
6. М.А.Грибков, А.В.Алексеевский, С.А.Спирин, М.А.Короткова Вычислительный подход к решению задачи о поиске максимальной клики [Электронный ресурс]. URL: http://www.isa.ru/proceedings/ images/documents/2006-25/185-192.pdf [Дата обращения 16 декабря 2017].
7. Савельев А.С. Алгоритмы поиска максимальной общей подструктуры набора молекул [Электронный ресурс]. URL: http://docplayer.ru/ 31550963-Algoritmy-poiska-maksimalnoy-obshchey-podstruktury-nabora-molekul-a-s-savelev-gga-software-services.html [Дата обращения 16 декабря 2017].
8. Гергель В.П. Высокопроизводительные вычисления для многоядерных многопроцессорных систем. – Учебное пособие – Нижний Новгород; Изд-во ННГУ им. Н.И. Лобачевского, 2010 [Электронный ресурс]. URL: http://www.unn.ru/pages/e-library/methodmaterial/2010/7.pdf [Дата обращения 16 декабря 2017].
9. Антонов А.С. Параллельное программирование с использованием технологии MPI: Учебное пособие. – M.: Изд-во МГУ, 2004. – 71 с.
10. Дискретная математика. Теория и практика решения задач по информатике [Электронный ресурс]: учебное пособие / С.М. Окулов – 2-е изд. (эл.). –М. : БИНОМ. Лаборатория знаний, 2012. – 422 с.

# ПРИЛОЖЕНИЕ А

Последовательная реализация алгоритма Брона-Кербоша на языке С++

void extend(list<int> cand,

list<int> not,

list<int> compsub,

const vector<list<int>> & lst,

list<list<int>> & SetCompsub,

int & Size)

{

while (!cand.empty() && Possible(cand, not, lst)) {

int v = cand.front();

compsub.push\_back(v);

list<int> new\_not = not;

list<int> new\_cands = cand

new\_not.remove\_if([v, lst](int l) {return (find(lst[v].cbegin(), lst[v].cend(), l) == lst[v].cend());});

new\_cands.remove\_if([v, lst](int l) {return (find(lst[v].cbegin(), lst[v].cend(), l) == lst[v].cend());});

if (new\_cands.empty() && new\_not.empty())

{

SetCompsub.push\_back(compsub);

if (compsub.size() > Size)

Size = compsub.size();

}

else extend(new\_cands, new\_not, compsub, lst, SetCompsub, Size);

compsub.remove(v);

not.push\_back(v);

cand.remove(v);

}

}

# ПРИЛОЖЕНИЕ Б

Текст процедуры, генерирующей случайный граф на для заданного количества вершин и рёбер

void Generate(int Ver, int Ed)

{

ofstream wr("graph" + to\_string(Ver) + "\_" + to\_string(Ed) + ".txt");

wr << Ver << endl;

int sum = 0;

vector<list<int>> vec (Ver);

uniform\_int\_distribution<int> uid (0, Ver-1);

random\_device rnd;

while (sum < Ed)

{

int a = uid(rnd);

int b = uid(rnd);

if (find(vec[a].begin(), vec[a].end(), b) == vec[a].end() && a != b)

{

//cout << a << " " << b << endl;

vec[a].push\_back(b);

vec[b].push\_back(a);

++sum;

wr << a+1 << " " << b+1 << endl;

}

}

wr.close();

}

# ПРИЛОЖЕНИЕ В

Текст процедуры, выполняющей статическое распределение вычислений для алгоритма Брона-Кербоша

void extend\_parallel(list<int> cand,

list<int> not,

list<int> compsub,

const vector<list<int>> & lst,

list<list<int>> & SetCompsub,

int & Size,

int num\_threads //количество потоков

)

{

int numb=0;

while (!cand.empty() && Possible(cand, not, lst))

{

int v = cand.front();//берём первый элемент

compsub.push\_back(v);

list<int> new\_not=not;

list<int> new\_cands=cand;

new\_not.remove\_if( [v, lst](int l) {return (find (lst[v].cbegin(), lst[v].cend(), l) == lst[v].cend());});

new\_cands.remove\_if([v, lst](int l) {return (find (lst[v].cbegin(), lst[v].cend(), l) == lst[v].cend());});

if (new\_cands.empty() && new\_not.empty())

{

if (omp\_get\_thread\_num == 0)

{

SetCompsub.push\_back(compsub);

if (compsub.size() > Size)

Size = compsub.size();

}

}

else

if (numb == omp\_get\_thread\_num())

{

extend(new\_cands, new\_not, compsub, lst, SetCompsub, Size);

}

numb = (numb + 1) % num\_threads;

compsub.remove(v);

not.push\_back(v);

cand.remove(v);

}

}

# ПРИЛОЖЕНИЕ Г

Текст процедуры, выполняющей статическое распределение вычислений для алгоритма Брона-Кербоша

int ClNum = 0;

void extend\_dynamic(list<int> cand,

list<int> not,

list<int> compsub,

const vector<list<int>> & lst,

list<list<int>> & SetCompsub,

int & Size

)

{

int numb = 0; //номер ветки

int flag = 0; //признак, запускать ветку или нет

while (!cand.empty() && Possible(cand, not, lst))

{

int v = cand.front();//берём первый элемент

compsub.push\_back(v);

list<int> new\_not = not;

list<int> new\_cands = cand;//из них надо исключить все вершины, не соединённые с v

new\_not.remove\_if([v, lst](int l) {return (find(lst[v].cbegin(), lst[v].cend(), l) == lst[v].cend());});//лямбда возвращает true, если ребра v-l не существует

new\_cands.remove\_if([v, lst](int l) {return (find(lst[v].cbegin(), lst[v].cend(), l) == lst[v].cend());});

if (new\_cands.empty() && new\_not.empty())//это означает, что compsub - это клика

{

if (omp\_get\_thread\_num == 0)

{

SetCompsub.push\_back(compsub);

if (compsub.size() > Size)

Size = compsub.size();

}

}

else

{

#pragma omp critical

{

if (numb==ClNum)//если эта ветка не запускалась

{

++ClNum;

flag = 1;

}

}

if (flag)

{

extend(new\_cands, new\_not, compsub, lst, SetCompsub, Size);

flag = 0;

}

}

++numb;

compsub.remove(v);

not.push\_back(v);

cand.remove(v);

}

}