Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа ТП.22Б07-мм

Экспериментальное исследование производительности алгоритма обхода графа в ширину

БУРАШНИКОВ Артем Максимович

Отчёт по учебной практике
в форме «Эксперимент»

Научный руководитель:
доцент кафедры информатики, к. ф.-м. н., С. В. Григорьев

Санкт-Петербург
2023

Оглавление

Введение **3**

1. Постановка задачи **4**

2. Обзор предметной области **5**

2.1. Существующие исследования 5

2.2. Обход в ширину в контексте линейной алгебры 6

3. Детали реализации **11**

3.1. Концепция алгоритма 11

3.2. Параллелизм 12

3.3. Реализация алгоритма на языке F# 13

4. Эксперимент **14**

4.1. Характеристики оборудования 14

4.2. Исследовательские вопросы 14

4.3. Использованные метрики 15

4.4. Набор данных 15

4.5. Постановка эксперимента 16

4.6. Анализ результатов 16

Заключение **17**

Список литературы **18**

Введение

Использование такой абстракции как *граф* для анализа и изучения различных форм реляционных данных имеет большое значение. Теоретические проблемы, существующие в областях применения, включают в себя определение и выявление значащих объектов, обнаружение аномалий, закономерностей или внезапных изменений, кластеризацию тесно связанных сущностей. Поиск в ширину (англ. — *Breadth-First Search*, сокр. — *BFS*) и поиск в глубину (англ. — *Depth-First Search*, сокр. — *DFS*) являются двумя основными алгоритмами для систематического исследование графов.

Для удовлетворения задач теоретического анализа в современных приложениях важна скорость вычислений. Одним из способов получить ускорение выступают *параллельные вычисления*, дающие выигрыш в производительности на современных многоядерных системах. Благодаря природе своей работы, алгоритм обхода в ширину естественным образом позволяёт внедрять в свою реализацию использование дополнительных *потоков* и *ядер процессора*. Таким образом, простота реализации, широкая область применения и возможность использования паралелльных вычислений делают BFS более популярным инструментом и объектом исследований, чем DFS.

1. Постановка задачи

Целью работы является провести экспериментальное исследование производительности обхода в ширину и ответить на следующие вопросы.

1. При каких параметрах графа выгоднее использовать параллельную версию алгоритма, а при каких последовательную?
2. Использование какого количества потоков даёт наибольший выигрыш в производительности и почему?

Для её выполнения были поставлены перечисленные ниже задачи.

- Реализовать параллельную и последовательную версии алгоритма обхода в ширину с использованием структур, подходящих для хранения в памяти компьютера разреженных матриц и векторов.
- Оценить влияние конкретных характеристик графа и количества используемых потоков на итоговую производительность BFS и найти критические величины этих параметров.

Выполнение поставленных задач позволит определить какая версия алгоритма (последовательная или параллельная) предпочтительнее к использованию при том или ином сценарии.

2. Обзор предметной области

Для проведения эксперимента необходимо ознакомиться с существующими работами, затрагивающими проблемы многопоточной реализации BFS. Кроме того, требуется рассмотреть преимущества использования абстрактных операций линейной алгебры в алгоритме обхода в ширину, отметив выбор подходящей структуры данных для хранения матриц и векторов, а также обратить внимание на особенности выбранного для реализации языка программирования.

2.1. Существующие исследования

Алгоритм обхода графа в ширину ввиду своей прикладной значимости был проанализирован в различном контексте в ряде исследовательских работ. Некоторые из них рассмотрены далее.

В работе [6] отмечается, что производительность BFS в значительной мере зависит от топологии подаваемого на вход графа. Ся Инлун (Xia Yinglong) и Празанна Виктор (Prasanna Viktor) показали, что в случае большого количества итераций алгоритма при малых количествах вершин/ребер между итерациями (то есть малом количестве вершин во фронте на каждой итерации) параллельная версия терпит снижение производительности из-за накладных расходов на создание параллельных задач.

Кроме того, в исследовании [4] Вират Агарвал (Virat Agarwal) и другие продемонстрировали, что последовательная версия алгоритма в некоторых ситуациях оказывается предпочтительнее не оптимизированной параллельной ввиду большой задержки работы с памятью и высокой вычислительной стоимостью её синхронизации.

Упомянутые авторы находят решение проблем оптимизации в тонкой настройке взаимодействия с общей памятью в используемой архитектуре или применении адаптивных алгоритмов, способных динамически контролировать количество используемых потоков во время исполнения.

Анализируя представленные работы, можно с высокой точностью прогнозировать зависимости между параметрами входных данных, вы-

бранной архитектурой и ожидаемой производительностью используемой реализации обхода в ширину. Данное экспериментальное исследование будет посвящено выявлению таких зависимостей для алгоритма BFS, реализованного с применением методов линейной алгебры, что существенным образом влияет не только на сам алгоритм, но и на внутреннее представление графа в памяти компьютера, в связи с чем полученные результаты могут представлять особый интерес.

2.2. Обход в ширину в контексте линейной алгебры

Для эффективного представления и манипулирования графовыми структурами можно использовать матрицы и вектора. Рассмотрим теоретические основы такой абстракции и её преимущества.

2.2.1. GraphBlas

Набор операций, которые могут быть применены к матрицам и векторам для выполнения различных графовых алгоритмов (таких как обходы, поиск путей, кластеризация и др.) заложен в *GraphBlas* [2] [1].

Это инициитива, нацеленная на создание стандартизованного интерфейса для разработки графовых алгоритмов. Её целью является обеспечение высокой производительности и переносимости для широкого спектра решений, работающих с различными графовыми структурами.

Пользователю предоставляется спецификация, определяющая абстрактное умножение матрицы на вектор, сложение векторов и умножение матрицы на матрицу. Описанная семантика и требования для интерфейса позволяют создавать собственные реализации на различных языках программирования и для различных аппаратных платформ.

2.2.2. Представление графа в памяти компьютера

Нужно отметить, что реальные графы сильно разреженны, но в то же время обладают сравнительно большим количеством вершин. Плотность

[1] Форум, посвещенный стандарту GraphBlas. Дата посещения: 23.05.2023

графа (англ. — *density*) можно вычислить по формуле:

$$\text{density} = \frac{2 \cdot |E|}{|V| \cdot (|V| - 1)}, \tag{1}$$

где $|E|$ — количество рёбер, а $|V|$ — количество вершин. Разреженными можно считать графы со значением плотности $< 10\%$. В то время как на практике в основном встречаются сильно разреженные графы со значением density $<<< 2\%$.

Хранить такие данные в виде двумерных таблиц не является эффективным решением, поэтому используют специальные структуры. Рассмотрим некоторые из них.

- **Список смежности**
  Это один из наиболее простых и эффективных способов хранения разреженных графов. Для каждой вершины выделяется список ее соседей. Это может быть реализовано с помощью массива списков, где каждый элемент массива представляет вершину, а связанный список содержит соседей этой вершины.

- **Матрица смежности**
  В матрице смежности каждый элемент указывает наличие или отсутствие ребра между двумя вершинами. В случае разреженных графов, где большая часть элементов матрицы будет нулевыми, может быть эффективно использована разреженная матрица, где хранятся только ненулевые значения.

- **Список ребер**
  Вместо хранения информации о соседних вершинах можно хранить список всех ребер графа. Каждое ребро тогда представляется парой вершин, которые оно соединяет. Этот подход эффективен для определенных операций, таких как перебор всех ребер.

- **Компактные структуры данных**
  Например, CSR (Compressed Sparse Row) и CSC (Compressed Sparse Column), которые оптимизируют использование памяти для разреженных графов, сохраняя при помощи массивов информацию о

вершинах, ребрах и их связях, и позволяют эффективно выполнять операции над разреженными графами.

Каждый метод имеет свои преимущества и недостатки с точки зрения памяти и производительности. Для сильно разреженных графов, которые были использованы для исследования BFS, в контексте линейной алгебры их представление лучше всего подходит в виде матрицы смежности. Более того, поскольку большинство значений у таких матриц равно нулю, для их хранения целесообрано использовать дополнительную структуру данных, абстракцией над которой выступают вектора и матрицы.

2.2.3. Деревья квадрантов

Деревья квадрантов (англ. — Quadtree) являются рекурсивной структурой данных, широко используемой для эффективного хранения и обработки разреженных матриц. Возможность представлять их компактным образом делает такие деревья ценным инструментом для оптимизации использования памяти.

Quadtree строится на основе деления матрицы смежности графа на четыре равные части (квадранта) и рекурсивного применения этого деления ко всем четырем дочерним квадрантам. Каждый такой квадрант может быть представлен в виде узла дерева, а в листьях хранится информация об объектах или данных, находящихся в соответствующем квадранте. Это позволяет эффективно представлять разреженные данные, так как области без объектов могут быть сохранены в упрощенной форме. На рисунке 1 представлена четверка узлов, образующих квадрант. Нумерацию и обозначение квадрантов принято вести слева направо, сверху вниз: (I) NW — North-West, (II) NE — North-East, (III) SW — South-West, (IV) SE — South-East

На рисунке 2 продемонстрирована ситуация, в которой на некотором уровне квадрант имеет всего один значащий узел, остальные элементы матрицы являются незначащими (в нашем случае для них использовано обозначение `None`).

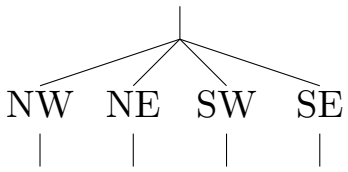

Рис. 1: Схематичное изображение одного из узлов дерева квадрантов и его потомков

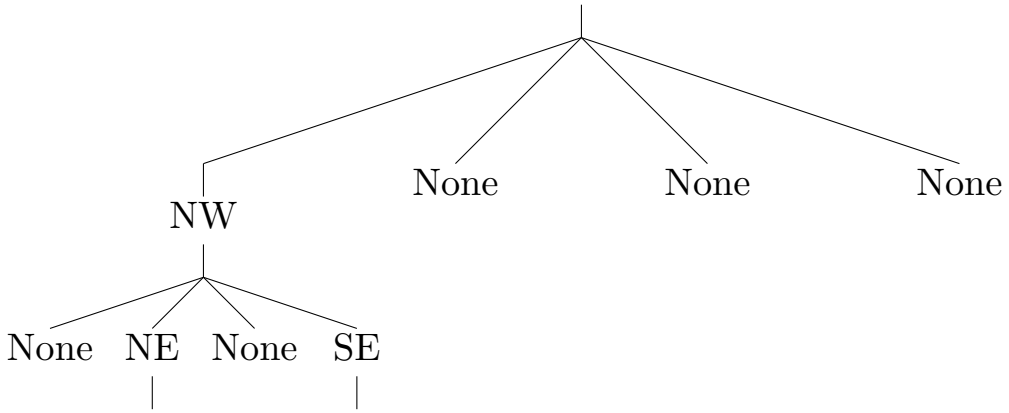

Рис. 2: Схематичное изображение узла дерева квадрантов только с одним потомков

Применение `None` для обозначения нулевых элементов не является случайным. Далее будут рассмотренны языковые особенности, позволившие естестевным образом внедрить обозаченные ранее абстракции и структуры

2.2.4. Языковые особенности `F#`

`Option` — это тип-контейнер, который может либо содержать значение определенного типа, либо быть пустым (`None`). Такое представление незначащих элементов очень удобно по нескольким причинам.

- **Естественное выражение отсутствия значения**
  Деревья квадрантов могут иметь узлы, которые не содержат никаких данных или не имеют потомков. Использование типа Option позволяет явно на это указать, делая код более читабельным.

- **Удобный матчинг (англ. — pattern-matching)**
  Эта особенность языка позволяет легко обрабатывать различные случаи, встречающиеся в узлах деревьев.

- **Безопасность типов**
  Использование `Option` обеспечивает статическую проверку типов. Например, компилятор предотвратит ошибки, связанные с попыткой обращения к значению там, где его нет.

3. Детали реализации

3.1. Концепция алгоритма

Основной принцип состоит в том, что применение операции умножения вектора на матрицу, определенной в GraphBlas, является переходом от одних вершин графа к другим по инцидентным этим вершинам рёбрам. В свою очередь вектора представлены в виде двоичного дерева, а матрицы — в виде дерева квадрантов. Такое представление удобно не только для хранения разреженных структур в памяти, но и для внедрения параллелизма. Рассмотрим конкретные шаги алгоритма.

- **Шаг 1**. Создается матрица смежности, соответствующая данному графу. В нашем случае по матрице смежности, представленной как список координат, строится дерево квадрантов, причем использование `F#` позволяет естественно выражать отсутствующие рёбра и отсутствующие метки через тип `Option`. Значения в ячейках матрицы заполняются значениями на соответствующих рёбрах графа.

- **Шаг 2**. Задаются два вектора.

    1. Фронт: вектор, в котором каждый индекс соответствует вершине графа, а значения указывают на текущие просматриваемые вершины. Отсутствие вершины в текущем фронте обозначается как `Option None`.

    2. Результирующий вектор, который при инициализации совпадает с фронтом а в процессе работы алгоритма аккумулирует промежуточные результаты (например, длину минимального пути).

- **Шаг 3**. Выполняется умножение вектора состояний на матрицу смежности. В качестве поэлементных операций сложения и умножения используются логические `ИЛИ` и `И`. После каждого умножения на результирующий вектор применяется маска, чтобы алгоритм

не обрабатывал уже посещенные вершины. Результатом итерации является новый фронт и обновленный результирующий вектор.

Шаг 3 повторяется до тех пор, пока не будут достигнуты требуемые условия остановки. Например, все вершины станут исследованы или произойдет выполнение определенного условия.

3.2. Параллелизм

Благодаря рекуррентной природе деревьев, являющихся в нашем случае внутренним представлением векторов и матриц, реализованный алгоритм естественным образом поддаётся распараллеливанию. В листинге 1 представлен псевдокод с использованием абстрактных операций линейной алгебры. *Минимальными единицами* такой операции являются вектор размера 1×2 и матрица размера 2×2, что на каждой итерации позволяет использовать до четырех потоков для разделения вычислений между ними.

$$\begin{bmatrix} a_{11} & a_{12} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} \times b_{11} + a_{12} \times b_{21} \\ a_{11} \times b_{12} + a_{12} \times b_{22} \end{bmatrix}$$

Элементы матрицы — поддеревья, поэтому вычисление элемента $a_{is} \times b_{sj}$ является рекурсивным вызовом на соответствующем поддереве. В представленной работе операция умножения над *минимальными единицами* разбивается на два потока: один из потоков отвечает за вычисление значения в первой строке результирующего вектора, а другой поток — за вычисление во второй. Такой подход выбран потому, что использовавшееся для постановки экспериментов оборудование имеет небольшое количество физических и виртуальных ядер процессора. Кроме того, при необходимости увеличить количество потоков на вход в BFS передаётся целый положительный параметр *parallelLevel*, величина которого уменьшается с каждым вызовом тела функции, в результате чего производятся дополнительные асинхронные вычисления на следующем уровне рекурсии. Этот передаваемый аргумент позволяет точно контролировать степень распараллеливания.

Листинг 1 Псевдокод параллельного алгоритма обхода в ширину с использованием методов линейной алгебры

```
function BFS (parallelLevel, startingVertices, adjacencyMatrix):

    matrix := makeSparseMatrix from adjacencyMatrix

    frontier := makeSparseVector from startVertices

    result := frontier

    def recursive function inner (frontier, result, counter):
        if frontier.IsEmpty then
            return result
        else
            newFrontier :=
                call multiplyVectorByMatrix (parallelLevel, frontier, matrix)

            newFrontier :=
                call applyMask (result, newFrontier)

            newResult :=
                call updateResult (count, result, newFrontier)

            newCounter := counter + 1

            call inner (newFrontier, newResult, newCounter)

    call inner (frontier, result, 1)
end function
```

3.3. Реализация алгоритма на языке F#

Ниже приведены ссылки на варианты исполнения обозначенных алгоритмов.

- Обход в ширину на языке `F#`[2].
- Умножения вектора на матрицу как абстракция над операциями над деревьями[3]

Отметим, что обе функции поддерживают как последовательное, так и параллельное исполнение.

[2]Это сноска

[3]

4. Эксперимент

4.1. Характеристики оборудования

Оборудование, на котором были поставлены описанные далее эксперименты, обладает следующими характеристиками:

Операционная система

```
Operating System: Ubuntu 22.04.2 LTS
```

CPU

```
Architecture:        x86_64
Model name:          AMD Ryzen 5 4500U with Radeon Graphics
Thread(s) per core: 1
Core(s) per socket: 6
Socket(s):           1
CPU max MHz:         2375,0000
CPU min MHz:         1400,0000
```

RAM

```
Total (MB): 9351
```

4.2. Исследовательские вопросы

Анализ поставленных задач позволил выдвинуть следующие гипотезы:

RQ1

Ожидается, что в параллельной версии алгоритма обхода в ширину производительность будет значительно превышать последовательную версию на сильно разреженных неориентированных графах, потому что

в таких графах большинство вершин имеют небольшую степень, что позволит эффективно распределить работу между потоками и уменьшить накладные расходы на синхронизацию. Таким образом, параллельная версия должна продемонстрировать ощутимое ускорение.

RQ2

Предполагается существование оптимального количества потоков в параллельной версии алгоритма, которое приведет к наибольшему выигрышу в производительности за счет эффективного использования доступных ресурсов вычислительной системы.

4.3. Использованные метрики

Для исследования RQ1 решено замерять ускорение (Speedup) параллельной версии алгоритма относительно последовательной со следующим набором контролируемых параметров:

- количество вершин в графе;
- плотность графа;
- количество используемых потоков.

Для поиска оптимального значения, обозначенного в RQ2, будет проанализировано среднее время работы параллельной версии алгоритма на сильно разреженных графах с использованием разного количества потоков: 1, 2, 4, 8, 16. Кроме того, зафиксируем распределение памяти и нагрузку на потоки во время исполнения. Все замеры будут выполнены с использованием библиотеки для измерения производительности `BenchmarkDotNet` [1], разрабатываемой и поддерживаемой для платформы `.NET`.

4.4. Набор данных

Для фиксации исследуемых величин были выбраны TODO различных разреженных квадратных матриц из коллекции университета Фло-

риды [5]. Плотные матрицы было решено генерировать. Необходимо учесть, чтобы матрица смежности графа заполнялась значениями меток равномерно по всей площади двумерной сетки, потому как группировка ребёр вокруг определенного квадранта матрицы может привести к нежелательным последствиям из-за особенности внутреннего представления матриц в виде деревьев. Для генерации матрицы смежности создавалась двумерная таблица, необходимое количество случайно выбранных ячеек которой заполнялось различными значениями. Функция выбирала ячейки с равномерным распределением.

Информация о выбранных данных представлена в таблице 1. Для обозначения числа ненулевых элементов используется аббревиатура *Nnz*. В таблице приведено название матрицы (при наличии — официальное), количество строк, количество ненулевых элементов, отношение ненулевых элементов к числу всех возможных элементов (разреженность).

Таблица 1: Разреженные матричные данные

Матрица	Количество строк R	Nnz M	Nnz$/R^2$

4.5. Постановка эксперимента

4.6. Анализ результатов

Заключение

Список литературы

[1] BenchmarkDotNet. — 0.13.4. URL: https://benchmarkdotnet.org/ (дата обращения: 15.05.2023).

[2] Mathematical foundations of the GraphBLAS / J. Kepner, P. Aaltonen, D. Bader et al. // 2016 IEEE High Performance Extreme Computing Conference (HPEC). — 2016. — P. 1–9.

[3] Newman M. E. J., Watts D. J., Strogatz S. H. Random graph models of social networks // Proceedings of the National Academy of Sciences. — 2002. — Vol. 99, no. suppl_1. — P. 2566–2572. — https://www.pnas.org/-doi/pdf/10.1073/pnas.012582999.

[4] Scalable Graph Exploration on Multicore Processors / Virat Agarwal, Fabrizio Petrini, Davide Pasetto, David A. Bader. — 2010.

[5] T. Davis. The SuiteSparse Matrix Collection (the University of Florida Sparse Matrix Collection). — 2020. — URL: https://sparse.tamu.edu (дата обращения: 15.05.2023).

[6] Xia Yinglong, Prasanna Viktor K. Topologically adaptive parallel breadth-first search on multicore processors. — 2009.