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

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. Решения для этих проблем обычно используют классические алгоритмы, применяемые для графов. Для удовлетворения потребностей теоретического анализа в современных приложениях важно ускорить работу алгоритмов, лежащих в основе этих задач. Одним из способов такого ускорения выступают *параллельные вычисления*, являющееся предпочтительной стратегией, дающей выигрыш в производительности на современных многоядерных системах.

Систематическое исследование всех ребер и вершин графа называется его обходом. Нужно отметить, что размеры графов, возникающих сегодня, массивны, но такие графы одновременно могут быть сильно разреженными, то есть иметь малое число рёбер по сравнению с количеством вершин. Для эффективного хранения таких объектов в памяти компьютера используются различные вспомогательные структуры, позволяющие существенно сэкономить занимаемое место.

Распараллеливание не всегда даёт существенный прирост производительности, а в каких-то случаях может значительно замедлить работу алгоритма. Поэтому интерес исследования представляют такие параметры параллельной реализации BFS и характеристики графов, для которых она оказывается эффективнее последовательной. Также отдельное влияние на производительность оказывают накладные расходы на создание самих параллельных задач, отвечающих за отдельные асинхронные вычисления. Указанное в совокупности определяет, какая версия (последовательная или параллельная) предпочтительнее к использованию при том или ином сценарии.

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 следующие задачи:

- Реализовать серию измерений производительности последовательной и параллельной версий обхода в ширину с контролируемым набором параметров графа, таких как общее количество вершин, разреженность графа и количество используемых вычислительных потоков.

- Оценить влияние конкретных характеристик графа на итоговую производительность BFS.

- Провести эксперименты с разным количеством потоков и измерить производительность алгоритма обхода в ширину при каждом сценарии, обращая внимание на накладные расходы при синхронизации и доступе к общей памяти.

Выполнение поставленных задач позволит связать заданные характеристики с эффективностью параллельного и последовательного алгоритма и обозначит границы параметров, при которых будет предпочтительна та или иная версия.

2. Сопутствующие исследования

Алгоритм обхода графа в ширину ввиду своей прикладной значимости был проанализирован в различном контексте в ряде исследовательских работ.

В [5] отмечается, что производительность BFS в значительной мере зависит от топологии подаваемого на вход графа. Авторы 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 существенное снижение производительности из-за накладных расходов.

Кроме того, в [3] продемонстрировано, что последовательная версия алгоритма во многих ситуациях оказывается предпочтительнее не оптимизированной параллельной ввиду большой задержки работы с памятью и высокой вычислительной стоимостью её синхронизации.

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

Обширный список проведенных исследований позволяет с высокой точностью прогнозировать зависимости между параметрами входных данных, выбранной архитектурой и ожидаемой производительностью используемой реализации обхода в ширину. Данная работа будет посвящена 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.

3. Детали реализации

3.1. GraphBlas

Обход в ширину был реализован на языке `F#` с использованием идей, заложенных `GraphBlas` [2].

`GraphBlas` - это инициатива, нацеленная на создание стандартизованного интерфейса для разработки графовых алгоритмов. Целью этой инициативы является обеспечение высокой производительности и переносимости для широкого спектра алгоритмов, работающих с различными графовыми структурами.

Пользователю предоставляется языково-независимая спецификация, определяющая набор абстрактных операций линейной алгебры: 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.

3.2. 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.

3.3. 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.

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

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
```

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.

3.4. Реализация алгоритма на языке F#

Ниже приведены ссылки на варианты исполнения обозначенных алгоритмов:

- Обход в ширину на языке `F#`.[1]
- Умножения вектора на матрицу как абстракция над операциями над деревьями.[2]

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

[1] https://github.com/artem-burashnikov/2022-Programming-Technologies-F-sharp-course/blob/fb0ba5a5acf9ab045d4e52da22681a9264ad5b32/src/spbu-fsharp/BreadthFirstSearch.fs

[2] https://github.com/artem-burashnikov/2022-Programming-Technologies-F-sharp-course/blob/fb0ba5a5acf9ab045d4e52da22681a9264ad5b32/src/spbu-fsharp/MatrixAlgebra.fs

4. Эксперимент

4.1. Характеристики оборудования

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

OS and Kernel

```
Operating System: Ubuntu 22.04.2 LTS
          Kernel: Linux 5.19.0-41-generic
```

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
L1d cache:           192 KiB (6 instances)
L1i cache:           192 KiB (6 instances)
L2 cache:            3 MiB (6 instances)
L3 cache:            8 MiB (2 instances)
```

GPU

```
Device: [AMD/ATI] Renoir (rev c3)
Memory: 256M
```

RAM

```
Memory:       7304
Swap memory: 2047
Total:        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 Флориды [4]. 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, соотношение ненулевых элементов к числу всех возможных элементов.

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] GraphBLAS Graph Linear Algebra API. — 2020. — URL: https://graphblas.github.io/ (дата обращения: 15.05.2023).

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

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

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